(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 3 859 519 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
04.08.2021 Bulletin 2021/31

(21) Application number: **21165109.6**

(22) Date of filing: **26.03.2018**

(51) Int Cl.:
G06F 7/483 *(2006.01)* ***G06F 7/544*** *(2006.01)*
G06F 9/30 *(2018.01)* ***G06N 3/04*** *(2006.01)*
G06N 3/063 *(2006.01)* ***G06N 3/08*** *(2006.01)*

(84) Designated Contracting States:
AL AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HR HU IE IS IT LI LT LU LV MC MK MT NL NO PL PT RO RS SE SI SK SM TR

(30) Priority: **28.04.2017 US 201762491699 P**
18.10.2017 US 201715787129

(62) Document number(s) of the earlier application(s) in accordance with Art. 76 EPC:
18164093.9 / 3 396 524

(71) Applicant: **INTEL Corporation**
Santa Clara, CA 95054 (US)

(72) Inventors:
- **Kaul, Himanshu**
  Portland, OR Oregon 97225 (US)
- **Anders, Mark A.**
  Hillsboro, OR Oregon 97124 (US)
- **Mathew, Sanu K.**
  Portland, OR Oregon 97229 (US)
- **Yao, Anbang**
  Beijing 11, 100190 (CN)
- **Ray, Joydeep**
  Folsom, CA California 95630 (US)
- **Tang, Ping T.**
  Edison, NJ New Jersey 08820 (US)
- **Strickland, Michael S.**
  Sunnyvale, CA California 94087 (US)
- **Chen, Xiaoming**
  Shanghai 31, 200241 (CN)
- **Shpeisman, Tatiana**
  Menlo Park, CA California 94025 (US)
- **Appu, Abhishek R.**
  El Dorado Hills, CA California 95762 (US)
- **Koker, Altug**
  El Dorado Hills, CA California 95762 (US)
- **Sinha, Kamal**
  Rancho Cordova, CA California 95742 (US)
- **Vembu, Balaji**
  Folsom, CA California 95630 (US)
- **Galoppo von Borries, Nicolas C.**
  Portland, OR Oregon 97212 (US)
- **Nurvitadhi, Eriko**
  Hillsboro, OR Oregon 97124 (US)
- **Barik, Rajkishore**
  Santa Clara, CA California 95054 (US)
- **Lin, Tsung-Han**
  Campbell, CA California 95008 (US)
- **Ranganathan, Vasanth**
  El Dorado Hills, CA California 95762 (US)
- **Jahagirdar, Sanjeev**
  Folsom, CA California 95630 (US)

(74) Representative: **Samson & Partner Patentanwälte mbB**
Widenmayerstraße 6
80538 München (DE)

Remarks:
This application was filed on 26-03-2021 as a divisional application to the application mentioned under INID code 62.

(54) INSTRUCTIONS AND LOGIC TO PERFORM FLOATING-POINT AND INTEGER OPERATIONS FOR MACHINE LEARNING

(57) The present disclosure provides an apparatus comprising a memory interface, an array of processing clusters each including a multiprocessor unit coupled to the memory interface, wherein at least one multiprocessor unit is to execute a fused multiply-add instruction in parallel across multiple threads. The at least one multiprocessor unit comprising a register file to store data, and a compute unit coupled to the register file, wherein the compute unit is to execute a fused multiply-add instruction on matrix data. The compute unit comprising hardware logic to quantize the data from a higher precision, including a 32-bit floating point format to a lower precision floating point format, including a 16-bit floating point format, having a 1-bit sign, an 8-bit exponent, and a mantissa, wherein fewer bits are used for the mantissa of the lower precision floating point format; and one or more logic units to perform the fused multiply-add operation on the data in the lower precision floating point format.

EP 3 859 519 A1

(Cont. next page)

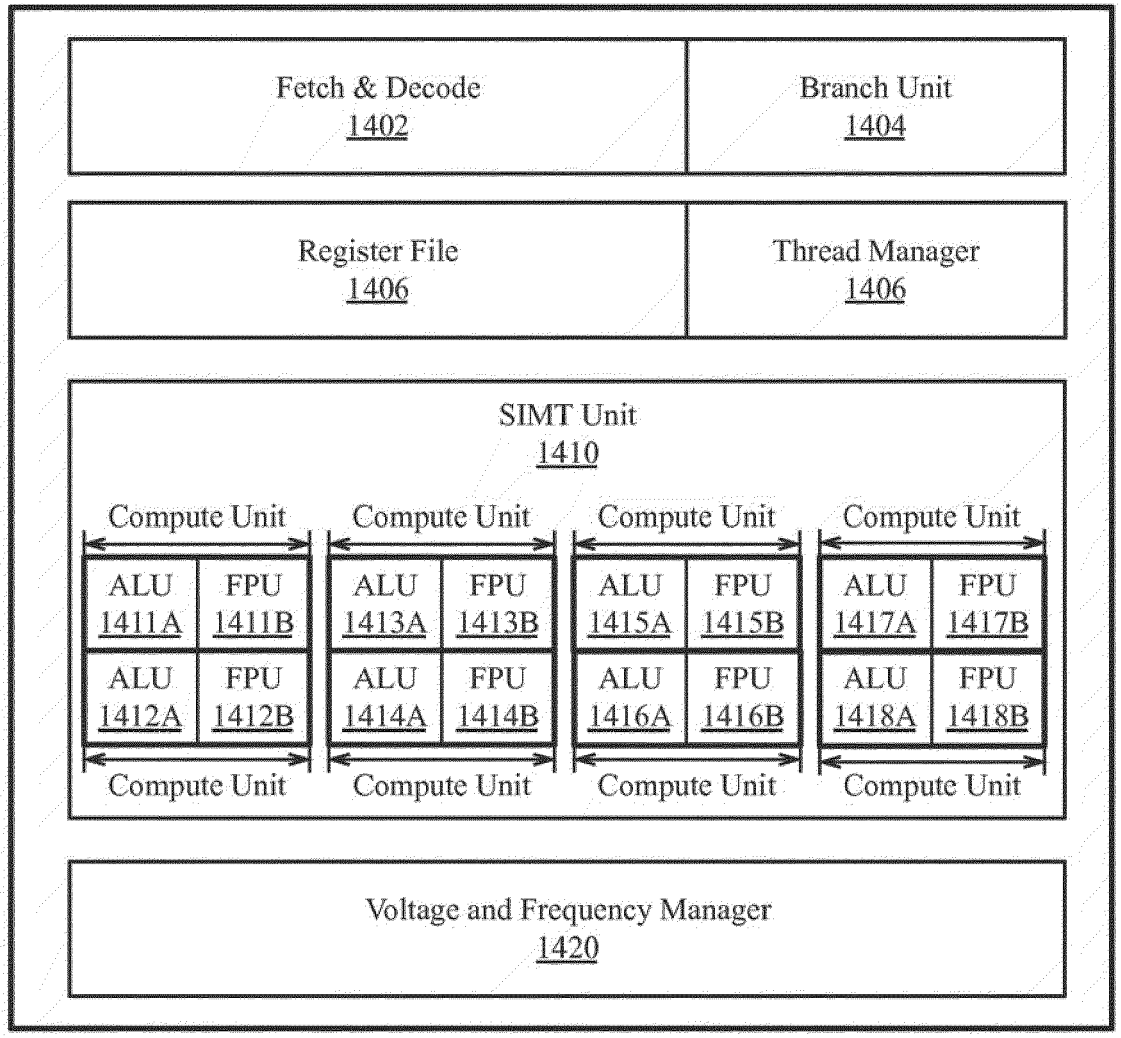

1400
Fetch & Decode
1402
Branch Unit
1404
Register File
1406
Thread Manager
1406
SIMT Unit
1410
Compute Unit
Compute Unit
Compute Unit
Compute Unit
ALU
1411A
FPU
1411B
ALU
1412A
FPU
1412B
ALU
1413A
FPU
1413B
ALU
1414A
FPU
1414B
ALU
1415A
FPU
1415B
ALU
1416A
FPU
1416B
ALU
1417A
FPU
1417B
ALU
1418A
FPU
1418B
Compute Unit
Compute Unit
Compute Unit
Compute Unit
Voltage and Frequency Manager
1420

FIG. 14

Description

FIELD

[0001] Embodiments relate generally to data processing and more particularly to data processing via a general-purpose graphics-processing unit.

BACKGROUND OF THE DESCRIPTION

[0002] Current parallel graphics data processing includes systems and methods developed to perform specific operations on graphics data such as, for example, linear interpolation, tessellation, rasterization, texture mapping, depth testing, etc. Traditionally, graphics processors used fixed function computational units to process graphics data; however, more recently, portions of graphics processors have been made programmable, enabling such processors to support a wider variety of operations for processing vertex and fragment data.

[0003] To further increase performance, graphics processors typically implement processing techniques such as pipelining that attempt to process, in parallel, as much graphics data as possible throughout the different parts of the graphics pipeline. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. A general overview of software and hardware for SIMT architectures can be found in Shane Cook, CUDA Programming, Chapter 3, pages 37-51 (2013) and/or Nicholas Wilt, CUDA Handbook, A Comprehensive Guide to GPU Programming, Sections 2.6.2 to 3.1.2 (June 2013).

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] So that the features of the present invention can be understood in detail, a more particular description of the invention may be had by reference to embodiments, some of which are illustrated in the appended drawings. It is to be noted, however, that the appended drawings illustrate only typical embodiments and are therefore not to be considered limiting of the scope of all embodiments.

FIG. 1 is a block diagram illustrating a computer system configured to implement one or more aspects of the embodiments described herein;
FIG. 2A-2D illustrate parallel processor components, according to an embodiment;
FIGs. 3A-3B are block diagrams of graphics multiprocessors, according to embodiments;
FIG. 4A-4F illustrate an exemplary architecture in which a plurality of GPUs is communicatively coupled to a plurality of multi-core processors;
FIG. 5 illustrates a graphics processing pipeline, according to an embodiment;
FIG. 6 illustrates a machine learning software stack, according to an embodiment;
FIG. 7 illustrates a highly-parallel general-purpose graphics processing unit, according to an embodiment;
FIG. 8 illustrates a multi-GPU computing system, according to an embodiment;
FIG. 9A-9B illustrate layers of exemplary deep neural networks;
FIG. 10 illustrates an exemplary recurrent neural network;
FIG. 11 illustrates training and deployment of a deep neural network;
FIG. 12 is a block diagram illustrating distributed learning;
FIG. 13 illustrates an exemplary inferencing system on a chip (SOC) suitable for performing inferencing using a trained model;
FIG. 14 is a block diagram of a multiprocessor unit, according to an embodiment;
FIG. 15A-15B illustrate designs for a logic units to perform integer and floating-point fused multiply-add operations, according to an embodiment;
FIG. 16 illustrates fused multiply-add logic unit having a merged floating-point and integer datapath, according to an embodiment;
FIG. 17A-17B illustrates logic units including merged computation circuits to perform floating point and integer fused-multiply accumulate operations, according to an embodiment;
FIG. 18A-18B illustrate a data processing system and associated compute and logic units that to perform accelerated training and inferencing operations for machine learning;
FIG. 19 illustrates details of the activation instruction module, according to an embodiment;
FIG. 20 illustrates the stochastic quantization unit, according to an embodiment;
FIG. 21 illustrates an FPU encoding and configuration module, according to one embodiment;

FIG. 22 illustrates logic to process an instruction using a dynamically configurable compute unit, according to an embodiment;
FIGs. 23A-23B are flow diagrams illustrating logic to perform sparse compute operations within a GPGPU provided by embodiments described herein;
FIG. 24 is a block diagram of a processing system, according to an embodiment;
FIG. 25 is a block diagram of a processor according to an embodiment;
FIG. 26 is a block diagram of a graphics processor, according to an embodiment;
FIG. 27 is a block diagram of a graphics processing engine of a graphics processor in accordance with some embodiments;
FIG. 28 is a block diagram of a graphics processor provided by an additional embodiment;
FIG. 29 illustrates thread execution logic including an array of processing elements employed in some embodiments;
FIG. 30 is a block diagram illustrating a graphics processor instruction formats according to some embodiments;
FIG. 31 is a block diagram of a graphics processor according to another embodiment.
FIG. 32A-32B illustrate a graphics processor command format and command sequence, according to some embodiments;
FIG. 33 illustrates exemplary graphics software architecture for a data processing system according to some embodiments;
FIG. 34 is a block diagram illustrating an IP core development system, according to an embodiment;
FIG. 35 is a block diagram illustrating an exemplary system on a chip integrated circuit, according to an embodiment;
FIG. 36 is a block diagram illustrating an additional graphics processor, according to an embodiment; and
FIG. 37 is a block diagram illustrating an additional exemplary graphics processor of a system on a chip integrated circuit, according to an embodiment.

DETAILED DESCRIPTION

[0005] In some embodiments, a graphics processing unit (GPU) is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or another interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

[0006] In the following description, numerous specific details are set forth to provide a more thorough understanding. However, it will be apparent to one of skill in the art that the embodiments described herein may be practiced without one or more of these specific details. In other instances, well-known features have not been described to avoid obscuring the details of the present embodiments.

System Overview

[0007] **FIG. 1** is a block diagram illustrating a computing system 100 configured to implement one or more aspects of the embodiments described herein. The computing system 100 includes a processing subsystem 101 having one or more processor(s) 102 and a system memory 104 communicating via an interconnection path that may include a memory hub 105. The memory hub 105 may be a separate component within a chipset component or may be integrated within the one or more processor(s) 102. The memory hub 105 couples with an I/O subsystem 111 via a communication link 106. The I/O subsystem 111 includes an I/O hub 107 that can enable the computing system 100 to receive input from one or more input device(s) 108. Additionally, the I/O hub 107 can enable a display controller, which may be included in the one or more processor(s) 102, to provide outputs to one or more display device(s) 110A. In one embodiment the one or more display device(s) 110A coupled with the I/O hub 107 can include a local, internal, or embedded display device.

[0008] In one embodiment the processing subsystem 101 includes one or more parallel processor(s) 112 coupled to memory hub 105 via a bus or other communication link 113. The communication link 113 may be one of any number of standards based communication link technologies or protocols, such as, but not limited to PCI Express, or may be a vendor specific communications interface or communications fabric. In one embodiment the one or more parallel processor(s) 112 form a computationally focused parallel or vector processing system that an include a large number of processing cores and/or processing clusters, such as a many integrated core (MIC) processor. In one embodiment the one or more parallel processor(s) 112 form a graphics processing subsystem that can output pixels to one of the one or more display device(s) 110A coupled via the I/O Hub 107. The one or more parallel processor(s) 112 can also include a display controller and display interface (not shown) to enable a direct connection to one or more display device(s) 110B.

[0009] Within the I/O subsystem 111, a system storage unit 114 can connect to the I/O hub 107 to provide a storage mechanism for the computing system 100. An I/O switch 116 can be used to provide an interface mechanism to enable connections between the I/O hub 107 and other components, such as a network adapter 118 and/or wireless network adapter 119 that may be integrated into the platform, and various other devices that can be added via one or more add-in device(s) 120. The network adapter 118 can be an Ethernet adapter or another wired network adapter. The wireless network adapter 119 can include one or more of a Wi-Fi, Bluetooth, near field communication (NFC), or other network device that includes one or more wireless radios.

[0010] The computing system 100 can include other components not explicitly shown, including USB or other port connections, optical storage drives, video capture devices, and the like, may also be connected to the I/O hub 107. Communication paths interconnecting the various components in FIG. 1 may be implemented using any suitable protocols, such as PCI (Peripheral Component Interconnect) based protocols (e.g., PCI-Express), or any other bus or point-to-point communication interfaces and/or protocol(s), such as the NV-Link high-speed interconnect, or interconnect protocols known in the art.

[0011] In one embodiment, the one or more parallel processor(s) 112 incorporate circuitry optimized for graphics and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). In another embodiment, the one or more parallel processor(s) 112 incorporate circuitry optimized for general purpose processing, while preserving the underlying computational architecture, described in greater detail herein. In yet another embodiment, components of the computing system 100 may be integrated with one or more other system elements on a single integrated circuit. For example, the one or more parallel processor(s), 112 memory hub 105, processor(s) 102, and I/O hub 107 can be integrated into a system on chip (SoC) integrated circuit. Alternatively, the components of the computing system 100 can be integrated into a single package to form a system in package (SIP) configuration. In one embodiment at least a portion of the components of the computing system 100 can be integrated into a multi-chip module (MCM), which can be interconnected with other multi-chip modules into a modular computing system.

[0012] It will be appreciated that the computing system 100 shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of processor(s) 102, and the number of parallel processor(s) 112, may be modified as desired. For instance, in some embodiments, system memory 104 is connected to the processor(s) 102 directly rather than through a bridge, while other devices communicate with system memory 104 via the memory hub 105 and the processor(s) 102. In other alternative topologies, the parallel processor(s) 112 are connected to the I/O hub 107 or directly to one of the one or more processor(s) 102, rather than to the memory hub 105. In other embodiments, the I/O hub 107 and memory hub 105 may be integrated into a single chip. Some embodiments may include two or more sets of processor(s) 102 attached via multiple sockets, which can couple with two or more instances of the parallel processor(s) 112.

[0013] Some of the particular components shown herein are optional and may not be included in all implementations of the computing system 100. For example, any number of add-in cards or peripherals may be supported, or some components may be eliminated. Furthermore, some architectures may use different terminology for components similar to those illustrated in FIG. 1. For example, the memory hub 105 may be referred to as a Northbridge in some architectures, while the I/O hub 107 may be referred to as a Southbridge.

[0014] **FIG. 2A** illustrates a parallel processor 200, according to an embodiment. The various components of the parallel processor 200 may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or field programmable gate arrays (FPGA). The illustrated parallel processor 200 is a variant of the one or more parallel processor(s) 112 shown in FIG. 1, according to an embodiment.

[0015] In one embodiment the parallel processor 200 includes a parallel processing unit 202. The parallel processing unit includes an I/O unit 204 that enables communication with other devices, including other instances of the parallel processing unit 202. The I/O unit 204 may be directly connected to other devices. In one embodiment the I/O unit 204 connects with other devices via the use of a hub or switch interface, such as memory hub 105. The connections between the memory hub 105 and the I/O unit 204 form a communication link 113. Within the parallel processing unit 202, the I/O unit 204 connects with a host interface 206 and a memory crossbar 216, where the host interface 206 receives commands directed to performing processing operations and the memory crossbar 216 receives commands directed to performing memory operations.

[0016] When the host interface 206 receives a command buffer via the I/O unit 204, the host interface 206 can direct work operations to perform those commands to a front end 208. In one embodiment the front end 208 couples with a scheduler 210, which is configured to distribute commands or other work items to a processing cluster array 212. In one embodiment the scheduler 210 ensures that the processing cluster array 212 is properly configured and in a valid state before tasks are distributed to the processing clusters of the processing cluster array 212. In one embodiment the scheduler 210 is implemented via firmware logic executing on a microcontroller. The microcontroller implemented scheduler 210 is configurable to perform complex scheduling and work distribution operations at coarse and fine granularity, enabling rapid preemption and context switching of threads executing on the processing array 212. In one embodiment, the host software can prove workloads for scheduling on the processing array 212 via one of multiple graphics processing

doorbells. The workloads can then be automatically distributed across the processing array 212 by the scheduler 210 logic within the scheduler microcontroller.

[0017] The processing cluster array 212 can include up to "N" processing clusters (e.g., cluster 214A, cluster 214B, through cluster 214N). Each cluster 214A-214N of the processing cluster array 212 can execute a large number of concurrent threads. The scheduler 210 can allocate work to the clusters 214A-214N of the processing cluster array 212 using various scheduling and/or work distribution algorithms, which may vary depending on the workload arising for each type of program or computation. The scheduling can be handled dynamically by the scheduler 210, or can be assisted in part by compiler logic during compilation of program logic configured for execution by the processing cluster array 212. In one embodiment, different clusters 214A-214N of the processing cluster array 212 can be allocated for processing different types of programs or for performing different types of computations.

[0018] The processing cluster array 212 can be configured to perform various types of parallel processing operations. In one embodiment the processing cluster array 212 is configured to perform general-purpose parallel compute operations. For example, the processing cluster array 212 can include logic to execute processing tasks including filtering of video and/or audio data, performing modeling operations, including physics operations, and performing data transformations.

[0019] In one embodiment the processing cluster array 212 is configured to perform parallel graphics processing operations. In embodiments in which the parallel processor 200 is configured to perform graphics processing operations, the processing cluster array 212 can include additional logic to support the execution of such graphics processing operations, including, but not limited to texture sampling logic to perform texture operations, as well as tessellation logic and other vertex processing logic. Additionally, the processing cluster array 212 can be configured to execute graphics processing related shader programs such as, but not limited to vertex shaders, tessellation shaders, geometry shaders, and pixel shaders. The parallel processing unit 202 can transfer data from system memory via the I/O unit 204 for processing. During processing the transferred data can be stored to on-chip memory (e.g., parallel processor memory 222) during processing, then written back to system memory.

[0020] In one embodiment, when the parallel processing unit 202 is used to perform graphics processing, the scheduler 210 can be configured to divide the processing workload into approximately equal sized tasks, to better enable distribution of the graphics processing operations to multiple clusters 214A-214N of the processing cluster array 212. In some embodiments, portions of the processing cluster array 212 can be configured to perform different types of processing. For example a first portion may be configured to perform vertex shading and topology generation, a second portion may be configured to perform tessellation and geometry shading, and a third portion may be configured to perform pixel shading or other screen space operations, to produce a rendered image for display. Intermediate data produced by one or more of the clusters 214A-214N may be stored in buffers to allow the intermediate data to be transmitted between clusters 214A-214N for further processing.

[0021] During operation, the processing cluster array 212 can receive processing tasks to be executed via the scheduler 210, which receives commands defining processing tasks from front end 208. For graphics processing operations, processing tasks can include indices of data to be processed, e.g., surface (patch) data, primitive data, vertex data, and/or pixel data, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The scheduler 210 may be configured to fetch the indices corresponding to the tasks or may receive the indices from the front end 208. The front end 208 can be configured to ensure the processing cluster array 212 is configured to a valid state before the workload specified by incoming command buffers (e.g., batch-buffers, push buffers, etc.) is initiated.

[0022] Each of the one or more instances of the parallel processing unit 202 can couple with parallel processor memory 222. The parallel processor memory 222 can be accessed via the memory crossbar 216, which can receive memory requests from the processing cluster array 212 as well as the I/O unit 204. The memory crossbar 216 can access the parallel processor memory 222 via a memory interface 218. The memory interface 218 can include multiple partition units (e.g., partition unit 220A, partition unit 220B, through partition unit 220N) that can each couple to a portion (e.g., memory unit) of parallel processor memory 222. In one implementation the number of partition units 220A-220N is configured to be equal to the number of memory units, such that a first partition unit 220A has a corresponding first memory unit 224A, a second partition unit 220B has a corresponding memory unit 224B, and an Nth partition unit 220N has a corresponding Nth memory unit 224N. In other embodiments, the number of partition units 220A-220N may not be equal to the number of memory devices.

[0023] In various embodiments, the memory units 224A-224N can include various types of memory devices, including dynamic random access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. In one embodiment, the memory units 224A-224N may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM). Persons skilled in the art will appreciate that the specific implementation of the memory units 224A-224N can vary, and can be selected from one of various conventional designs. Render targets, such as frame buffers or texture maps may be stored across the memory units 224A-224N, allowing partition units 220A-220N to write portions of each render target in parallel

to efficiently use the available bandwidth of parallel processor memory 222. In some embodiments, a local instance of the parallel processor memory 222 may be excluded in favor of a unified memory design that utilizes system memory in conjunction with local cache memory.

[0024] In one embodiment, any one of the clusters 214A-214N of the processing cluster array 212 can process data that will be written to any of the memory units 224A-224N within parallel processor memory 222. The memory crossbar 216 can be configured to transfer the output of each cluster 214A-214N to any partition unit 220A-220N or to another cluster 214A-214N, which can perform additional processing operations on the output. Each cluster 214A-214N can communicate with the memory interface 218 through the memory crossbar 216 to read from or write to various external memory devices. In one embodiment the memory crossbar 216 has a connection to the memory interface 218 to communicate with the I/O unit 204, as well as a connection to a local instance of the parallel processor memory 222, enabling the processing units within the different processing clusters 214A-214N to communicate with system memory or other memory that is not local to the parallel processing unit 202. In one embodiment the memory crossbar 216 can use virtual channels to separate traffic streams between the clusters 214A-214N and the partition units 220A-220N.

[0025] While a single instance of the parallel processing unit 202 is illustrated within the parallel processor 200, any number of instances of the parallel processing unit 202 can be included. For example, multiple instances of the parallel processing unit 202 can be provided on a single add-in card, or multiple add-in cards can be interconnected. The different instances of the parallel processing unit 202 can be configured to inter-operate even if the different instances have different numbers of processing cores, different amounts of local parallel processor memory, and/or other configuration differences. For example and in one embodiment, some instances of the parallel processing unit 202 can include higher precision floating point units relative to other instances. Systems incorporating one or more instances of the parallel processing unit 202 or the parallel processor 200 can be implemented in a variety of configurations and form factors, including but not limited to desktop, laptop, or handheld personal computers, servers, workstations, game consoles, and/or embedded systems.

[0026] **FIG. 2B** is a block diagram of a partition unit 220, according to an embodiment. In one embodiment the partition unit 220 is an instance of one of the partition units 220A-220N of FIG. 2A. As illustrated, the partition unit 220 includes an L2 cache 221, a frame buffer interface 225, and a ROP 226 (raster operations unit). The L2 cache 221 is a read/write cache that is configured to perform load and store operations received from the memory crossbar 216 and ROP 226. Read misses and urgent write-back requests are output by L2 cache 221 to frame buffer interface 225 for processing. Updates can also be sent to the frame buffer via the frame buffer interface 225 for processing. In one embodiment the frame buffer interface 225 interfaces with one of the memory units in parallel processor memory, such as the memory units 224A-224N of FIG. 2 (e.g., within parallel processor memory 222).

[0027] In graphics applications, the ROP 226 is a processing unit that performs raster operations such as stencil, z test, blending, and the like. The ROP 226 then outputs processed graphics data that is stored in graphics memory. In some embodiments the ROP 226 includes compression logic to compress depth or color data that is written to memory and decompress depth or color data that is read from memory. The compression logic can be lossless compression logic that makes use of one or more of multiple compression algorithms. The type of compression that is performed by the ROP 226 can vary based on the statistical characteristics of the data to be compressed. For example, in one embodiment, delta color compression is performed on depth and color data on a per-tile basis.

[0028] In some embodiments, the ROP 226 is included within each processing cluster (e.g., cluster 214A-214N of FIG. 2) instead of within the partition unit 220. In such embodiment, read and write requests for pixel data are transmitted over the memory crossbar 216 instead of pixel fragment data. The processed graphics data may be displayed on a display device, such as one of the one or more display device(s) 110 of FIG. 1, routed for further processing by the processor(s) 102, or routed for further processing by one of the processing entities within the parallel processor 200 of FIG. 2A.

[0029] **FIG. 2C** is a block diagram of a processing cluster 214 within a parallel processing unit, according to an embodiment. In one embodiment the processing cluster is an instance of one of the processing clusters 214A-214N of FIG. 2. The processing cluster 214 can be configured to execute many threads in parallel, where the term "thread" refers to an instance of a particular program executing on a particular set of input data. In some embodiments, single-instruction, multiple-data (SIMD) instruction issue techniques are used to support parallel execution of a large number of threads without providing multiple independent instruction units. In other embodiments, single-instruction, multiple-thread (SIMT) techniques are used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the processing clusters. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons skilled in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.

[0030] Operation of the processing cluster 214 can be controlled via a pipeline manager 232 that distributes processing tasks to SIMT parallel processors. The pipeline manager 232 receives instructions from the scheduler 210 of FIG. 2 and manages execution of those instructions via a graphics multiprocessor 234 and/or a texture unit 236. The illustrated

graphics multiprocessor 234 is an exemplary instance of a SIMT parallel processor. However, various types of SIMT parallel processors of differing architectures may be included within the processing cluster 214. One or more instances of the graphics multiprocessor 234 can be included within a processing cluster 214. The graphics multiprocessor 234 can process data and a data crossbar 240 can be used to distribute the processed data to one of multiple possible destinations, including other shader units. The pipeline manager 232 can facilitate the distribution of processed data by specifying destinations for processed data to be distributed vis the data crossbar 240.

[0031] Each graphics multiprocessor 234 within the processing cluster 214 can include an identical set of functional execution logic (e.g., arithmetic logic units, load-store units, etc.). The functional execution logic can be configured in a pipelined manner in which new instructions can be issued before previous instructions are complete. The functional execution logic supports a variety of operations including integer and floating point arithmetic, comparison operations, Boolean operations, bit-shifting, and computation of various algebraic functions. In one embodiment the same functional-unit hardware can be leveraged to perform different operations and any combination of functional units may be present.

[0032] The instructions transmitted to the processing cluster 214 constitutes a thread. A set of threads executing across the set of parallel processing engines is a thread group. A thread group executes the same program on different input data. Each thread within a thread group can be assigned to a different processing engine within a graphics multiprocessor 234. A thread group may include fewer threads than the number of processing engines within the graphics multiprocessor 234. When a thread group includes fewer threads than the number of processing engines, one or more of the processing engines may be idle during cycles in which that thread group is being processed. A thread group may also include more threads than the number of processing engines within the graphics multiprocessor 234. When the thread group includes more threads than the number of processing engines within the graphics multiprocessor 234, processing can be performed over consecutive clock cycles. In one embodiment multiple thread groups can be executed concurrently on a graphics multiprocessor 234.

[0033] In one embodiment the graphics multiprocessor 234 includes an internal cache memory to perform load and store operations. In one embodiment, the graphics multiprocessor 234 can forego an internal cache and use a cache memory (e.g., L1 cache 308) within the processing cluster 214. Each graphics multiprocessor 234 also has access to L2 caches within the partition units (e.g., partition units 220A-220N of FIG. 2) that are shared among all processing clusters 214 and may be used to transfer data between threads. The graphics multiprocessor 234 may also access off-chip global memory, which can include one or more of local parallel processor memory and/or system memory. Any memory external to the parallel processing unit 202 may be used as global memory. Embodiments in which the processing cluster 214 includes multiple instances of the graphics multiprocessor 234 can share common instructions and data, which may be stored in the L1 cache 308.

[0034] Each processing cluster 214 may include an MMU 245 (memory management unit) that is configured to map virtual addresses into physical addresses. In other embodiments, one or more instances of the MMU 245 may reside within the memory interface 218 of FIG. 2. The MMU 245 includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile and optionally a cache line index. The MMU 245 may include address translation lookaside buffers (TLB) or caches that may reside within the graphics multiprocessor 234 or the L1 cache or processing cluster 214. The physical address is processed to distribute surface data access locality to allow efficient request interleaving among partition units. The cache line index may be used to determine whether a request for a cache line is a hit or miss.

[0035] In graphics and computing applications, a processing cluster 214 may be configured such that each graphics multiprocessor 234 is coupled to a texture unit 236 for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within graphics multiprocessor 234 and is fetched from an L2 cache, local parallel processor memory, or system memory, as needed. Each graphics multiprocessor 234 outputs processed tasks to the data crossbar 240 to provide the processed task to another processing cluster 214 for further processing or to store the processed task in an L2 cache, local parallel processor memory, or system memory via the memory crossbar 216. A preROP 242 (pre-raster operations unit) is configured to receive data from graphics multiprocessor 234, direct data to ROP units, which may be located with partition units as described herein (e.g., partition units 220A-220N of FIG. 2). The preROP 242 unit can perform optimizations for color blending, organize pixel color data, and perform address translations.

[0036] It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., graphics multiprocessor 234, texture units 236, preROPs 242, etc., may be included within a processing cluster 214. Further, while only one processing cluster 214 is shown, a parallel processing unit as described herein may include any number of instances of the processing cluster 214. In one embodiment, each processing cluster 214 can be configured to operate independently of other processing clusters 214 using separate and distinct processing units, L1 caches, etc.

[0037] FIG. 2D shows a graphics multiprocessor 234, according to one embodiment. In such embodiment the graphics multiprocessor 234 couples with the pipeline manager 232 of the processing cluster 214. The graphics multiprocessor

234 has an execution pipeline including but not limited to an instruction cache 252, an instruction unit 254, an address mapping unit 256, a register file 258, one or more general purpose graphics processing unit (GPGPU) cores 262, and one or more load/store units 266. The GPGPU cores 262 and load/store units 266 are coupled with cache memory 272 and shared memory 270 via a memory and cache interconnect 268.

[0038] In one embodiment, the instruction cache 252 receives a stream of instructions to execute from the pipeline manager 232. The instructions are cached in the instruction cache 252 and dispatched for execution by the instruction unit 254. The instruction unit 254 can dispatch instructions as thread groups (e.g., warps), with each thread of the thread group assigned to a different execution unit within GPGPU core 262. An instruction can access any of a local, shared, or global address space by specifying an address within a unified address space. The address mapping unit 256 can be used to translate addresses in the unified address space into a distinct memory address that can be accessed by the load/store units 266.

[0039] The register file 258 provides a set of registers for the functional units of the graphics multiprocessor 324. The register file 258 provides temporary storage for operands connected to the data paths of the functional units (e.g., GPGPU cores 262, load/store units 266) of the graphics multiprocessor 324. In one embodiment, the register file 258 is divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file 258. In one embodiment, the register file 258 is divided between the different warps being executed by the graphics multiprocessor 324.

[0040] The GPGPU cores 262 can each include floating point units (FPUs) and/or integer arithmetic logic units (ALUs) that are used to execute instructions of the graphics multiprocessor 324. The GPGPU cores 262 can be similar in architecture or can differ in architecture, according to embodiments. For example and in one embodiment, a first portion of the GPGPU cores 262 include a single precision FPU and an integer ALU while a second portion of the GPGPU cores include a double precision FPU. In one embodiment the FPUs can implement the IEEE 754-2008 standard for floating point arithmetic or enable variable precision floating point arithmetic. The graphics multiprocessor 324 can additionally include one or more fixed function or special function units to perform specific functions such as copy rectangle or pixel blending operations. In one embodiment one or more of the GPGPU cores can also include fixed or special function logic.

[0041] In one embodiment the GPGPU cores 262 include SIMD logic capable of performing a single instruction on multiple sets of data. In one embodiment GPGPU cores 262 can physically execute SIMD4, SIMD8, and SIMD16 instructions and logically execute SIMD1, SIMD2, and SIMD32 instructions. The SIMD instructions for the GPGPU cores can be generated at compile time by a shader compiler or automatically generated when executing programs written and compiled for single program multiple data (SPMD) or SIMT architectures. Multiple threads of a program configured for the SIMT execution model can executed via a single SIMD instruction. For example and in one embodiment, eight SIMT threads that perform the same or similar operations can be executed in parallel via a single SIMD8 logic unit.

[0042] The memory and cache interconnect 268 is an interconnect network that connects each of the functional units of the graphics multiprocessor 324 to the register file 258 and to the shared memory 270. In one embodiment, the memory and cache interconnect 268 is a crossbar interconnect that allows the load/store unit 266 to implement load and store operations between the shared memory 270 and the register file 258. The register file 258 can operate at the same frequency as the GPGPU cores 262, thus data transfer between the GPGPU cores 262 and the register file 258 is very low latency. The shared memory 270 can be used to enable communication between threads that execute on the functional units within the graphics multiprocessor 234. The cache memory 272 can be used as a data cache for example, to cache texture data communicated between the functional units and the texture unit 236. The shared memory 270 can also be used as a program managed cached. Threads executing on the GPGPU cores 262 can programmatically store data within the shared memory in addition to the automatically cached data that is stored within the cache memory 272.

[0043] **FIGs. 3A-3B** illustrate additional graphics multiprocessors, according to embodiments. The illustrated graphics multiprocessors 325, 350 are variants of the graphics multiprocessor 234 of FIG. 2C. The illustrated graphics multiprocessors 325, 350 can be configured as a streaming multiprocessor (SM) capable of simultaneous execution of a large number of execution threads.

[0044] FIG. 3A shows a graphics multiprocessor 325 according to an additional embodiment. The graphics multiprocessor 325 includes multiple additional instances of execution resource units relative to the graphics multiprocessor 234 of FIG. 2D. For example, the graphics multiprocessor 325 can include multiple instances of the instruction unit 332A-332B, register file 334A-334B, and texture unit(s) 344A-344B. The graphics multiprocessor 325 also includes multiple sets of graphics or compute execution units (e.g., GPGPU core 336A-336B, GPGPU core 337A-337B, GPGPU core 338A-338B) and multiple sets of load/store units 340A-340B. In one embodiment the execution resource units have a common instruction cache 330, texture and/or data cache memory 342, and shared memory 346.

[0045] The various components can communicate via an interconnect fabric 327. In one embodiment the interconnect fabric 327 includes one or more crossbar switches to enable communication between the various components of the graphics multiprocessor 325. In one embodiment the interconnect fabric 327 is a separate, high-speed network fabric layer upon which each component of the graphics multiprocessor 325 is stacked. The components of the graphics

multiprocessor 325 communicate with remote components via the interconnect fabric 327. For example, the GPGPU cores 336A-336B, 337A-337B, and 3378A-338B can each communicate with shared memory 346 via the interconnect fabric 327. The interconnect fabric 327 can arbitrate communication within the graphics multiprocessor 325 to ensure a fair bandwidth allocation between components.

[0046] FIG. 3B shows a graphics multiprocessor 350 according to an additional embodiment. The graphics processor includes multiple sets of execution resources 356A-356D, where each set of execution resource includes multiple instruction units, register files, GPGPU cores, and load store units, as illustrated in FIG. 2D and FIG. 3A. The execution resources 356A-356D can work in concert with texture unit(s) 360A-360D for texture operations, while sharing an instruction cache 354, and shared memory 362. In one embodiment the execution resources 356A-356D can share an instruction cache 354 and shared memory 362, as well as multiple instances of a texture and/or data cache memory 358A-358B. The various components can communicate via an interconnect fabric 352 similar to the interconnect fabric 327 of FIG. 3A.

[0047] Persons skilled in the art will understand that the architecture described in FIGS. 1, 2A-2D, and 3A-3B are descriptive and not limiting as to the scope of the present embodiments. Thus, the techniques described herein may be implemented on any properly configured processing unit, including, without limitation, one or more mobile application processors, one or more desktop or server central processing units (CPUs) including multi-core CPUs, one or more parallel processing units, such as the parallel processing unit 202 of FIG. 2, as well as one or more graphics processors or special purpose processing units, without departure from the scope of the embodiments described herein.

[0048] In some embodiments a parallel processor or GPGPU as described herein is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or other interconnect (e.g., a high speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

Techniques for GPU to Host Processor Interconnection

[0049] **FIG. 4A** illustrates an exemplary architecture in which a plurality of GPUs 410-413 are communicatively coupled to a plurality of multi-core processors 405-406 over high-speed links 440-443 (e.g., buses, point-to-point interconnects, etc.). In one embodiment, the high-speed links 440-443 support a communication throughput of 4GB/s, 30GB/s, 80GB/s or higher, depending on the implementation. Various interconnect protocols may be used including, but not limited to, PCIe 4.0 or 5.0 and NVLink 2.0. However, the underlying principles of the invention are not limited to any particular communication protocol or throughput.

[0050] In addition, in one embodiment, two or more of the GPUs 410-413 are interconnected over high-speed links 444-445, which may be implemented using the same or different protocols/links than those used for high-speed links 440-443. Similarly, two or more of the multi-core processors 405-406 may be connected over high speed link 433 which may be symmetric multi-processor (SMP) buses operating at 20GB/s, 30GB/s, 120GB/s or higher. Alternatively, all communication between the various system components shown in FIG. 4A may be accomplished using the same protocols/links (e.g., over a common interconnection fabric). As mentioned, however, the underlying principles of the invention are not limited to any particular type of interconnect technology.

[0051] In one embodiment, each multi-core processor 405-406 is communicatively coupled to a processor memory 401-402, via memory interconnects 430-431, respectively, and each GPU 410-413 is communicatively coupled to GPU memory 420-423 over GPU memory interconnects 450-453, respectively. The memory interconnects 430-431 and 450-453 may utilize the same or different memory access technologies. By way of example, and not limitation, the processor memories 401-402 and GPU memories 420-423 may be volatile memories such as dynamic random access memories (DRAMs) (including stacked DRAMs), Graphics DDR SDRAM (GDDR) (e.g., GDDR5, GDDR6), or High Bandwidth Memory (HBM) and/or may be non-volatile memories such as 3D XPoint or Nano-RAM. In one embodiment, some portion of the memories may be volatile memory and another portion may be non-volatile memory (e.g., using a two-level memory (2LM) hierarchy).

[0052] As described below, although the various processors 405-406 and GPUs 410-413 may be physically coupled to a particular memory 401-402, 420-423, respectively, a unified memory architecture may be implemented in which the same virtual system address space (also referred to as the "effective address" space) is distributed among all of the various physical memories. For example, processor memories 401-402 may each comprise 64GB of the system memory address space and GPU memories 420-423 may each comprise 32GB of the system memory address space (resulting in a total of 256GB addressable memory in this example).

[0053] **FIG. 4B** illustrates additional details for an interconnection between a multi-core processor 407 and a graphics acceleration module 446 in accordance with one embodiment. The graphics acceleration module 446 may include one or more GPU chips integrated on a line card which is coupled to the processor 407 via the high-speed link 440. Alternatively, the graphics acceleration module 446 may be integrated on the same package or chip as the processor 407.

[0054] The illustrated processor 407 includes a plurality of cores 460A-460D, each with a translation lookaside buffer 461A-461D and one or more caches 462A-462D. The cores may include various other components for executing instructions and processing data which are not illustrated to avoid obscuring the underlying principles of the invention (e.g., instruction fetch units, branch prediction units, decoders, execution units, reorder buffers, etc.). The caches 462A-462D may comprise level 1 (LI) and level 2 (L2) caches. In addition, one or more shared caches 426 may be included in the caching hierarchy and shared by sets of the cores 460A-460D. For example, one embodiment of the processor 407 includes 24 cores, each with its own L1 cache, twelve shared L2 caches, and twelve shared L3 caches. In this embodiment, one of the L2 and L3 caches are shared by two adjacent cores. The processor 407 and the graphics accelerator integration module 446 connect with system memory 441, which may include processor memories 401-402

[0055] Coherency is maintained for data and instructions stored in the various caches 462A-462D, 456 and system memory 441 via inter-core communication over a coherence bus 464. For example, each cache may have cache coherency logic/circuitry associated therewith to communicate to over the coherence bus 464 in response to detected reads or writes to particular cache lines. In one implementation, a cache snooping protocol is implemented over the coherence bus 464 to snoop cache accesses. Cache snooping/coherency techniques are well understood by those of skill in the art and will not be described in detail here to avoid obscuring the underlying principles of the invention.

[0056] In one embodiment, a proxy circuit 425 communicatively couples the graphics acceleration module 446 to the coherence bus 464, allowing the graphics acceleration module 446 to participate in the cache coherence protocol as a peer of the cores. In particular, an interface 435 provides connectivity to the proxy circuit 425 over high-speed link 440 (e.g., a PCIe bus, NVLink, etc.) and an interface 437 connects the graphics acceleration module 446 to the high-speed link 440.

[0057] In one implementation, an accelerator integration circuit 436 provides cache management, memory access, context management, and interrupt management services on behalf of a plurality of graphics processing engines 431, 432, N of the graphics acceleration module 446. The graphics processing engines 431, 432, N may each comprise a separate graphics processing unit (GPU). Alternatively, the graphics processing engines 431, 432, N may comprise different types of graphics processing engines within a GPU such as graphics execution units, media processing engines (e.g., video encoders/decoders), samplers, and blit engines. In other words, the graphics acceleration module may be a GPU with a plurality of graphics processing engines 431-432, N or the graphics processing engines 431-432, N may be individual GPUs integrated on a common package, line card, or chip.

[0058] In one embodiment, the accelerator integration circuit 436 includes a memory management unit (MMU) 439 for performing various memory management functions such as virtual-to-physical memory translations (also referred to as effective-to-real memory translations) and memory access protocols for accessing system memory 441. The MMU 439 may also include a translation lookaside buffer (TLB) (not shown) for caching the virtual/effective to physical/real address translations. In one implementation, a cache 438 stores commands and data for efficient access by the graphics processing engines 431-432, N. In one embodiment, the data stored in cache 438 and graphics memories 433-434, N is kept coherent with the core caches 462A-462D, 456 and system memory 411. As mentioned, this may be accomplished via proxy circuit 425 which takes part in the cache coherency mechanism on behalf of cache 438 and memories 433-434, N (e.g., sending updates to the cache 438 related to modifications/accesses of cache lines on processor caches 462A-462D, 456 and receiving updates from the cache 438).

[0059] A set of registers 449 store context data for threads executed by the graphics processing engines 431-432, N and a context management circuit 448 manages the thread contexts. For example, the context management circuit 448 may perform save and restore operations to save and restore contexts of the various threads during contexts switches (e.g., where a first thread is saved and a second thread is stored so that the second thread can be execute by a graphics processing engine). For example, on a context switch, the context management circuit 448 may store current register values to a designated region in memory (e.g., identified by a context pointer). It may then restore the register values when returning to the context. In one embodiment, an interrupt management circuit 447 receives and processes interrupts received from system devices.

[0060] In one implementation, virtual/effective addresses from a graphics processing engine 431 are translated to real/physical addresses in system memory 411 by the MMU 439. One embodiment of the accelerator integration circuit 436 supports multiple (e.g., 4, 8, 16) graphics accelerator modules 446 and/or other accelerator devices. The graphics accelerator module 446 may be dedicated to a single application executed on the processor 407 or may be shared between multiple applications. In one embodiment, a virtualized graphics execution environment is presented in which the resources of the graphics processing engines 431-432, N are shared with multiple applications or virtual machines (VMs). The resources may be subdivided into "slices" which are allocated to different VMs and/or applications based on the processing requirements and priorities associated with the VMs and/or applications.

[0061] Thus, the accelerator integration circuit acts as a bridge to the system for the graphics acceleration module 446 and provides address translation and system memory cache services. In addition, the accelerator integration circuit 436 may provide virtualization facilities for the host processor to manage virtualization of the graphics processing engines, interrupts, and memory management.

[0062] Because hardware resources of the graphics processing engines 431-432, N are mapped explicitly to the real address space seen by the host processor 407, any host processor can address these resources directly using an effective address value. One function of the accelerator integration circuit 436, in one embodiment, is the physical separation of the graphics processing engines 431-432, N so that they appear to the system as independent units.

[0063] As mentioned, in the illustrated embodiment, one or more graphics memories 433-434, M are coupled to each of the graphics processing engines 431-432, N, respectively. The graphics memories 433-434, M store instructions and data being processed by each of the graphics processing engines 431-432, N. The graphics memories 433-434, M may be volatile memories such as DRAMs (including stacked DRAMs), GDDR memory (e.g., GDDR5, GDDR6), or HBM, and/or may be non-volatile memories such as 3D XPoint or Nano-Ram.

[0064] In one embodiment, to reduce data traffic over the high-speed link 440, biasing techniques are used to ensure that the data stored in graphics memories 433-434, M is data which will be used most frequently by the graphics processing engines 431-432, N and preferably not used by the cores 460A-460D (at least not frequently). Similarly, the biasing mechanism attempts to keep data needed by the cores (and preferably not the graphics processing engines 431-432, N) within the caches 462A-462D, 456 of the cores and system memory 411.

[0065] **FIG. 4C** illustrates another embodiment in which the accelerator integration circuit 436 is integrated within the processor 407. In this embodiment, the graphics processing engines 431-432, N communicate directly over the high-speed link 440 to the accelerator integration circuit 436 via interface 437 and interface 435 (which, again, may be utilize any form of bus or interface protocol). The accelerator integration circuit 436 may perform the same operations as those described with respect to **FIG. 4B,** but potentially at a higher throughput given its close proximity to the coherency bus 462 and caches 462A-462D, 426.

[0066] One embodiment supports different programming models including a dedicated-process programming model (no graphics acceleration module virtualization) and shared programming models (with virtualization). The latter may include programming models which are controlled by the accelerator integration circuit 436 and programming models which are controlled by the graphics acceleration module 446.

[0067] In one embodiment of the dedicated process model, graphics processing engines 431-432, N are dedicated to a single application or process under a single operating system. The single application can funnel other application requests to the graphics engines 431-432, N, providing virtualization within a VM/partition.

[0068] In the dedicated-process programming models, the graphics processing engines 431-432, N, may be shared by multiple VM/application partitions. The shared models require a system hypervisor to virtualize the graphics processing engines 431-432, N to allow access by each operating system. For single-partition systems without a hypervisor, the graphics processing engines 431-432, N are owned by the operating system. In both cases, the operating system can virtualize the graphics processing engines 431-432, N to provide access to each process or application.

[0069] For the shared programming model, the graphics acceleration module 446 or an individual graphics processing engine 431-432, N selects a process element using a process handle. In one embodiment, process elements are stored in system memory 411 and are addressable using the effective address to real address translation techniques described herein. The process handle may be an implementation-specific value provided to the host process when registering its context with the graphics processing engine 431-432, N (that is, calling system software to add the process element to the process element linked list). The lower 16-bits of the process handle may be the offset of the process element within the process element linked list.

[0070] **FIG. 4D** illustrates an exemplary accelerator integration slice 490. As used herein, a "slice" comprises a specified portion of the processing resources of the accelerator integration circuit 436. Application effective address space 482 within system memory 411 stores process elements 483. In one embodiment, the process elements 483 are stored in response to GPU invocations 481 from applications 480 executed on the processor 407. A process element 483 contains the process state for the corresponding application 480. A work descriptor (WD) 484 contained in the process element 483 can be a single job requested by an application or may contain a pointer to a queue of jobs. In the latter case, the WD 484 is a pointer to the job request queue in the application's address space 482.

[0071] The graphics acceleration module 446 and/or the individual graphics processing engines 431-432, N can be shared by all or a subset of the processes in the system. Embodiments of the invention include an infrastructure for setting up the process state and sending a WD 484 to a graphics acceleration module 446 to start a job in a virtualized environment.

[0072] In one implementation, the dedicated-process programming model is implementation-specific. In this model, a single process owns the graphics acceleration module 446 or an individual graphics processing engine 431. Because the graphics acceleration module 446 is owned by a single process, the hypervisor initializes the accelerator integration circuit 436 for the owning partition and the operating system initializes the accelerator integration circuit 436 for the

owning process at the time when the graphics acceleration module 446 is assigned.

[0073] In operation, a WD fetch unit 491 in the accelerator integration slice 490 fetches the next WD 484 which includes an indication of the work to be done by one of the graphics processing engines of the graphics acceleration module 446. Data from the WD 484 may be stored in registers 449 and used by the MMU 439, interrupt management circuit 447 and/or context management circuit 446 as illustrated. For example, one embodiment of the MMU 439 includes segment/page walk circuitry for accessing segment/page tables 486 within the OS virtual address space 485. The interrupt management circuit 447 may process interrupt events 492 received from the graphics acceleration module 446. When performing graphics operations, an effective address 493 generated by a graphics processing engine 431-432, N is translated to a real address by the MMU 439.

[0074] In one embodiment, the same set of registers 449 are duplicated for each graphics processing engine 431-432, N and/or graphics acceleration module 446 and may be initialized by the hypervisor or operating system. Each of these duplicated registers may be included in an accelerator integration slice 490. Exemplary registers that may be initialized by the hypervisor are shown in **Table 1.**

Table 1 - Hypervisor Initialized Registers

1	Slice Control Register
2	Real Address (RA) Scheduled Processes Area Pointer
3	Authority Mask Override Register
4	Interrupt Vector Table Entry Offset
5	Interrupt Vector Table Entry Limit
6	State Register
7	Logical Partition ID
8	Real address (RA) Hypervisor Accelerator Utilization Record Pointer
9	Storage Description Register

[0075] Exemplary registers that may be initialized by the operating system are shown in **Table 2.**

Table 2 - Operating System Initialized Registers

1	Process and Thread Identification
2	Effective Address (EA) Context Save/Restore Pointer
3	Virtual Address (VA) Accelerator Utilization Record Pointer
4	Virtual Address (VA) Storage Segment Table Pointer
5	Authority Mask
6	Work descriptor

[0076] In one embodiment, each WD 484 is specific to a particular graphics acceleration module 446 and/or graphics processing engine 431-432, N. It contains all the information a graphics processing engine 431-432, N requires to do its work or it can be a pointer to a memory location where the application has set up a command queue of work to be completed.

[0077] **FIG. 4E** illustrates additional details for one embodiment of a shared model. This embodiment includes a hypervisor real address space 498 in which a process element list 499 is stored. The hypervisor real address space 498 is accessible via a hypervisor 496 which virtualizes the graphics acceleration module engines for the operating system 495.

[0078] The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use a graphics acceleration module 446. There are two programming models where the graphics acceleration module 446 is shared by multiple processes and partitions: time-sliced shared and graphics directed shared.

[0079] In this model, the system hypervisor 496 owns the graphics acceleration module 446 and makes its function available to all operating systems 495. For a graphics acceleration module 446 to support virtualization by the system hypervisor 496, the graphics acceleration module 446 may adhere to the following requirements: 1) An application's job request must be autonomous (that is, the state does not need to be maintained between jobs), or the graphics acceleration

module 446 must provide a context save and restore mechanism. 2) An application's job request is guaranteed by the graphics acceleration module 446 to complete in a specified amount of time, including any translation faults, or the graphics acceleration module 446 provides the ability to preempt the processing of the job. 3) The graphics acceleration module 446 must be guaranteed fairness between processes when operating in the directed shared programming model.

[0080] In one embodiment, for the shared model, the application 480 is required to make an operating system 495 system call with a graphics acceleration module 446 type, a work descriptor (WD), an authority mask register (AMR) value, and a context save/restore area pointer (CSRP). The graphics acceleration module 446 type describes the targeted acceleration function for the system call. The graphics acceleration module 446 type may be a system-specific value. The WD is formatted specifically for the graphics acceleration module 446 and can be in the form of a graphics acceleration module 446 command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the graphics acceleration module 446. In one embodiment, the AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR. If the accelerator integration circuit 436 and graphics acceleration module 446 implementations do not support a User Authority Mask Override Register (UAMOR), the operating system may apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call. The hypervisor 496 may optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element 483. In one embodiment, the CSRP is one of the registers 449 containing the effective address of an area in the application's address space 482 for the graphics acceleration module 446 to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area may be pinned system memory.

[0081] Upon receiving the system call, the operating system 495 may verify that the application 480 has registered and been given the authority to use the graphics acceleration module 446. The operating system 495 then calls the hypervisor 496 with the information shown in **Table 3.**

Table 3 - OS to Hypervisor Call Parameters

1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)

[0082] Upon receiving the hypervisor call, the hypervisor 496 verifies that the operating system 495 has registered and been given the authority to use the graphics acceleration module 446. The hypervisor 496 then puts the process element 483 into the process element linked list for the corresponding graphics acceleration module 446 type. The process element may include the information shown in **Table 4.**

Table 4 - Process Element Information

1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)
8	Interrupt vector table, derived from the hypervisor call parameters.
9	A state register (SR) value
10	A logical partition ID (LPID)

(continued)

11	A real address (RA) hypervisor accelerator utilization record pointer
12	The Storage Descriptor Register (SDR)

[0083] In one embodiment, the hypervisor initializes a plurality of accelerator integration slice 490 registers 449.

[0084] As illustrated in **FIG. 4F,** one embodiment of the invention employs a unified memory addressable via a common virtual memory address space used to access the physical processor memories 401-402 and GPU memories 420-423. In this implementation, operations executed on the GPUs 410-413 utilize the same virtual/effective memory address space to access the processors memories 401-402 and vice versa, thereby simplifying programmability. In one embodiment, a first portion of the virtual/effective address space is allocated to the processor memory 401, a second portion to the second processor memory 402, a third portion to the GPU memory 420, and so on. The entire virtual/effective memory space (sometimes referred to as the effective address space) is thereby distributed across each of the processor memories 401-402 and GPU memories 420-423, allowing any processor or GPU to access any physical memory with a virtual address mapped to that memory.

[0085] In one embodiment, bias/coherence management circuitry 494A-494E within one or more of the MMUs 439A-439E ensures cache coherence between the caches of the host processors (e.g., 405) and the GPUs 410-413 and implements biasing techniques indicating the physical memories in which certain types of data should be stored. While multiple instances of bias/coherence management circuitry 494A-494E are illustrated in FIG. 4F, the bias/coherence circuitry may be implemented within the MMU of one or more host processors 405 and/or within the accelerator integration circuit 436.

[0086] One embodiment allows GPU-attached memory 420-423 to be mapped as part of system memory, and accessed using shared virtual memory (SVM) technology, but without suffering the typical performance drawbacks associated with full system cache coherence. The ability to GPU-attached memory 420-423 to be accessed as system memory without onerous cache coherence overhead provides a beneficial operating environment for GPU offload. This arrangement allows the host processor 405 software to setup operands and access computation results, without the overhead of tradition I/O direct memory access (DMA) data copies. Such traditional copies involve driver calls, interrupts and memory mapped I/O (MMIO) accesses that are all inefficient relative to simple memory accesses. At the same time, the ability to access GPU attached memory 420-423 without cache coherence overheads can be critical to the execution time of an offloaded computation. In cases with substantial streaming write memory traffic, for example, cache coherence overhead can significantly reduce the effective write bandwidth seen by a GPU 410-413. The efficiency of operand setup, the efficiency of results access, and the efficiency of GPU computation all play a role in determining the effectiveness of GPU offload.

[0087] In one implementation, the selection of between GPU bias and host processor bias is driven by a bias tracker data structure. A bias table may be used, for example, which may be a page-granular structure (i.e., controlled at the granularity of a memory page) that includes 1 or 2 bits per GPU-attached memory page. The bias table may be implemented in a stolen memory range of one or more GPU-attached memories 420-423, with or without a bias cache in the GPU 410-413 (e.g., to cache frequently/recently used entries of the bias table). Alternatively, the entire bias table may be maintained within the GPU.

[0088] In one implementation, the bias table entry associated with each access to the GPU-attached memory 420-423 is accessed prior the actual access to the GPU memory, causing the following operations. First, local requests from the GPU 410-413 that find their page in GPU bias are forwarded directly to a corresponding GPU memory 420-423. Local requests from the GPU that find their page in host bias are forwarded to the processor 405 (e.g., over a high-speed link as discussed above). In one embodiment, requests from the processor 405 that find the requested page in host processor bias complete the request like a normal memory read. Alternatively, requests directed to a GPU-biased page may be forwarded to the GPU 410-413. The GPU may then transition the page to a host processor bias if it is not currently using the page.

[0089] The bias state of a page can be changed either by a software-based mechanism, a hardware-assisted software-based mechanism, or, for a limited set of cases, a purely hardware-based mechanism.

[0090] One mechanism for changing the bias state employs an API call (e.g. OpenCL), which, in turn, calls the GPU's device driver which, in turn, sends a message (or enqueues a command descriptor) to the GPU directing it to change the bias state and, for some transitions, perform a cache flushing operation in the host. The cache flushing operation is required for a transition from host processor 405 bias to GPU bias, but is not required for the opposite transition.

[0091] In one embodiment, cache coherency is maintained by temporarily rendering GPU-biased pages uncacheable by the host processor 405. To access these pages, the processor 405 may request access from the GPU 410 which may or may not grant access right away, depending on the implementation. Thus, to reduce communication between the processor 405 and GPU 410 it is beneficial to ensure that GPU-biased pages are those which are required by the

GPU but not the host processor 405 and vice versa.

Graphics Processing Pipeline

[0092] **FIG. 5** illustrates a graphics processing pipeline 500, according to an embodiment. In one embodiment a graphics processor can implement the illustrated graphics processing pipeline 500. The graphics processor can be included within the parallel processing subsystems as described herein, such as the parallel processor 200 of FIG. 2, which, in one embodiment, is a variant of the parallel processor(s) 112 of FIG. 1. The various parallel processing systems can implement the graphics processing pipeline 500 via one or more instances of the parallel processing unit (e.g., parallel processing unit 202 of FIG. 2) as described herein. For example, a shader unit (e.g., graphics multiprocessor 234 of FIG. 3) may be configured to perform the functions of one or more of a vertex processing unit 504, a tessellation control processing unit 508, a tessellation evaluation processing unit 512, a geometry processing unit 516, and a fragment/pixel processing unit 524. The functions of data assembler 502, primitive assemblers 506, 514, 518, tessellation unit 510, rasterizer 522, and raster operations unit 526 may also be performed by other processing engines within a processing cluster (e.g., processing cluster 214 of FIG. 3) and a corresponding partition unit (e.g., partition unit 220A-220N of FIG. 2). The graphics processing pipeline 500 may also be implemented using dedicated processing units for one or more functions. In one embodiment, one or more portions of the graphics processing pipeline 500 can be performed by parallel processing logic within a general purpose processor (e.g., CPU). In one embodiment, one or more portions of the graphics processing pipeline 500 can access on-chip memory (e.g., parallel processor memory 222 as in FIG. 2) via a memory interface 528, which may be an instance of the memory interface 218 of FIG. 2.

[0093] In one embodiment the data assembler 502 is a processing unit that collects vertex data for surfaces and primitives. The data assembler 502 then outputs the vertex data, including the vertex attributes, to the vertex processing unit 504. The vertex processing unit 504 is a programmable execution unit that executes vertex shader programs, lighting and transforming vertex data as specified by the vertex shader programs. The vertex processing unit 504 reads data that is stored in cache, local or system memory for use in processing the vertex data and may be programmed to transform the vertex data from an object-based coordinate representation to a world space coordinate space or a normalized device coordinate space.

[0094] A first instance of a primitive assembler 506 receives vertex attributes from the vertex processing unit 50. The primitive assembler 506 readings stored vertex attributes as needed and constructs graphics primitives for processing by tessellation control processing unit 508. The graphics primitives include triangles, line segments, points, patches, and so forth, as supported by various graphics processing application programming interfaces (APIs).

[0095] The tessellation control processing unit 508 treats the input vertices as control points for a geometric patch. The control points are transformed from an input representation from the patch (e.g., the patch's bases) to a representation that is suitable for use in surface evaluation by the tessellation evaluation processing unit 512. The tessellation control processing unit 508 can also compute tessellation factors for edges of geometric patches. A tessellation factor applies to a single edge and quantifies a view-dependent level of detail associated with the edge. A tessellation unit 510 is configured to receive the tessellation factors for edges of a patch and to tessellate the patch into multiple geometric primitives such as line, triangle, or quadrilateral primitives, which are transmitted to a tessellation evaluation processing unit 512. The tessellation evaluation processing unit 512 operates on parameterized coordinates of the subdivided patch to generate a surface representation and vertex attributes for each vertex associated with the geometric primitives.

[0096] A second instance of a primitive assembler 514 receives vertex attributes from the tessellation evaluation processing unit 512, reading stored vertex attributes as needed, and constructs graphics primitives for processing by the geometry processing unit 516. The geometry processing unit 516 is a programmable execution unit that executes geometry shader programs to transform graphics primitives received from primitive assembler 514 as specified by the geometry shader programs. In one embodiment the geometry processing unit 516 is programmed to subdivide the graphics primitives into one or more new graphics primitives and calculate parameters used to rasterize the new graphics primitives.

[0097] In some embodiments the geometry processing unit 516 can add or delete elements in the geometry stream. The geometry processing unit 516 outputs the parameters and vertices specifying new graphics primitives to primitive assembler 518. The primitive assembler 518 receives the parameters and vertices from the geometry processing unit 516 and constructs graphics primitives for processing by a viewport scale, cull, and clip unit 520. The geometry processing unit 516 reads data that is stored in parallel processor memory or system memory for use in processing the geometry data. The viewport scale, cull, and clip unit 520 performs clipping, culling, and viewport scaling and outputs processed graphics primitives to a rasterizer 522.

[0098] The rasterizer 522 can perform depth culling and other depth-based optimizations. The rasterizer 522 also performs scan conversion on the new graphics primitives to generate fragments and output those fragments and associated coverage data to the fragment/pixel processing unit 524. The fragment/pixel processing unit 524 is a programmable execution unit that is configured to execute fragment shader programs or pixel shader programs. The fragment/pixel

processing unit 524 transforming fragments or pixels received from rasterizer 522, as specified by the fragment or pixel shader programs. For example, the fragment/pixel processing unit 524 may be programmed to perform operations included but not limited to texture mapping, shading, blending, texture correction and perspective correction to produce shaded fragments or pixels that are output to a raster operations unit 526. The fragment/pixel processing unit 524 can read data that is stored in either the parallel processor memory or the system memory for use when processing the fragment data. Fragment or pixel shader programs may be configured to shade at sample, pixel, tile, or other granularities depending on the sampling rate configured for the processing units.

[0099] The raster operations unit 526 is a processing unit that performs raster operations including, but not limited to stencil, z test, blending, and the like, and outputs pixel data as processed graphics data to be stored in graphics memory (e.g., parallel processor memory 222 as in FIG. 2, and/or system memory 104 as in FIG 1, to be displayed on the one or more display device(s) 110 or for further processing by one of the one or more processor(s) 102 or parallel processor(s) 112. In some embodiments the raster operations unit 526 is configured to compress z or color data that is written to memory and decompress z or color data that is read from memory.

Machine Learning Overview

[0100] A machine learning algorithm is an algorithm that can learn based on a set of data. Embodiments of machine learning algorithms can be designed to model high-level abstractions within a data set. For example, image recognition algorithms can be used to determine which of several categories to which a given input belong; regression algorithms can output a numerical value given an input; and pattern recognition algorithms can be used to generate translated text or perform text to speech and/or speech recognition.

[0101] An exemplary type of machine learning algorithm is a neural network. There are many types of neural networks; a simple type of neural network is a feedforward network. A feedforward network may be implemented as an acyclic graph in which the nodes are arranged in layers. Typically, a feedforward network topology includes an input layer and an output layer that are separated by at least one hidden layer. The hidden layer transforms input received by the input layer into a representation that is useful for generating output in the output layer. The network nodes are fully connected via edges to the nodes in adjacent layers, but there are no edges between nodes within each layer. Data received at the nodes of an input layer of a feedforward network are propagated (i.e., "fed forward") to the nodes of the output layer via an activation function that calculates the states of the nodes of each successive layer in the network based on coefficients ("weights") respectively associated with each of the edges connecting the layers. Depending on the specific model being represented by the algorithm being executed, the output from the neural network algorithm can take various forms.

[0102] Before a machine learning algorithm can be used to model a particular problem, the algorithm is trained using a training data set. Training a neural network involves selecting a network topology, using a set of training data representing a problem being modeled by the network, and adjusting the weights until the network model performs with a minimal error for all instances of the training data set. For example, during a supervised learning training process for a neural network, the output produced by the network in response to the input representing an instance in a training data set is compared to the "correct" labeled output for that instance, an error signal representing the difference between the output and the labeled output is calculated, and the weights associated with the connections are adjusted to minimize that error as the error signal is backward propagated through the layers of the network. The network is considered "trained" when the errors for each of the outputs generated from the instances of the training data set are minimized.

[0103] The accuracy of a machine learning algorithm can be affected significantly by the quality of the data set used to train the algorithm. The training process can be computationally intensive and may require a significant amount of time on a conventional general-purpose processor. Accordingly, parallel processing hardware is used to train many types of machine learning algorithms. This is particularly useful for optimizing the training of neural networks, as the computations performed in adjusting the coefficients in neural networks lend themselves naturally to parallel implementations. Specifically, many machine learning algorithms and software applications have been adapted to make use of the parallel processing hardware within general-purpose graphics processing devices.

[0104] **FIG. 6** is a generalized diagram of a machine learning software stack 600. A machine learning application 602 can be configured to train a neural network using a training dataset or to use a trained deep neural network to implement machine intelligence. The machine learning application 602 can include training and inference functionality for a neural network and/or specialized software that can be used to train a neural network before deployment. The machine learning application 602 can implement any type of machine intelligence including but not limited to image recognition, mapping and localization, autonomous navigation, speech synthesis, medical imaging, or language translation.

[0105] Hardware acceleration for the machine learning application 602 can be enabled via a machine learning framework 604. The machine learning framework 604 can provide a library of machine learning primitives. Machine learning primitives are basic operations that are commonly performed by machine learning algorithms. Without the machine learning framework 604, developers of machine learning algorithms would be required to create and optimize the main

computational logic associated with the machine learning algorithm, then re-optimize the computational logic as new parallel processors are developed. Instead, the machine learning application can be configured to perform the necessary computations using the primitives provided by the machine learning framework 604. Exemplary primitives include tensor convolutions, activation functions, and pooling, which are computational operations that are performed while training a convolutional neural network (CNN). The machine learning framework 604 can also provide primitives to implement basic linear algebra subprograms performed by many machine-learning algorithms, such as matrix and vector operations.

[0106] The machine learning framework 604 can process input data received from the machine learning application 602 and generate the appropriate input to a compute framework 606. The compute framework 606 can abstract the underlying instructions provided to the GPGPU driver 608 to enable the machine learning framework 604 to take advantage of hardware acceleration via the GPGPU hardware 610 without requiring the machine learning framework 604 to have intimate knowledge of the architecture of the GPGPU hardware 610. Additionally, the compute framework 606 can enable hardware acceleration for the machine learning framework 604 across a variety of types and generations of the GPGPU hardware 610.

GPGPU Machine Learning Acceleration

[0107] **FIG. 7** illustrates a highly-parallel general-purpose graphics processing unit 700, according to an embodiment. In one embodiment the general-purpose processing unit (GPGPU) 700 can be configured to be particularly efficient in processing the type of computational workloads associated with training deep neural networks. Additionally, the GPGPU 700 can be linked directly to other instances of the GPGPU to create a multi-GPU cluster to improve training speed for particularly deep neural networks.

[0108] The GPGPU 700 includes a host interface 702 to enable a connection with a host processor. In one embodiment the host interface 702 is a PCI Express interface. However, the host interface can also be a vendor specific communications interface or communications fabric. The GPGPU 700 receives commands from the host processor and uses a global scheduler 704 to distribute execution threads associated with those commands to a set of compute clusters 706A-706H. The compute clusters 706A-706H share a cache memory 708. The cache memory 708 can serve as a higher-level cache for cache memories within the compute clusters 706A-706H.

[0109] The GPGPU 700 includes memory 714A-714B coupled with the compute clusters 706A-H via a set of memory controllers 712A-712B. In various embodiments, the memory 714A-714B can include various types of memory devices including dynamic random access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory, or 3D stacked memory, including but not limited to high bandwidth memory (HBM).

[0110] In one embodiment each compute cluster 706A-706H includes a set of graphics multiprocessors, such as the graphics multiprocessor 400 of FIG. 4A. The graphics multiprocessors of the compute cluster multiple types of integer and floating point logic units that can perform computational operations at a range of precisions including suited for machine learning computations. For example and in one embodiment at least a subset of the floating point units in each of the compute clusters 706A-706H can be configured to perform 16-bit or 32-bit floating point operations, while a different subset of the floating point units can be configured to perform 64-bit floating point operations.

[0111] Multiple instances of the GPGPU 700 can be configured to operate as a compute cluster. The communication mechanism used by the compute cluster for synchronization and data exchange varies across embodiments. In one embodiment the multiple instances of the GPGPU 700 communicate over the host interface 702. In one embodiment the GPGPU 700 includes an I/O hub 709 that couples the GPGPU 700 with a GPU link 710 that enables a direct connection to other instances of the GPGPU. In one embodiment the GPU link 710 is coupled to a dedicated GPU-to-GPU bridge that enables communication and synchronization between multiple instances of the GPGPU 700. In one embodiment the GPU link 710 couples with a high speed interconnect to transmit and receive data to other GPGPUs or parallel processors. In one embodiment the multiple instances of the GPGPU 700 are located in separate data processing systems and communicate via a network device that is accessible via the host interface 702. In one embodiment the GPU link 710 can be configured to enable a connection to a host processor in addition to or as an alternative to the host interface 702.

[0112] While the illustrated configuration of the GPGPU 700 can be configured to train neural networks, one embodiment provides alternate configuration of the GPGPU 700 that can be configured for deployment within a high performance or low power inferencing platform. In an inferencing configuration the GPGPU 700 includes fewer of the compute clusters 706A-706H relative to the training configuration. Additionally memory technology associated with the memory 714A-714B may differ between inferencing and training configurations. In one embodiment the inferencing configuration of the GPGPU 700 can support inferencing specific instructions. For example, an inferencing configuration can provide support for one or more 8-bit integer dot product instructions, which are commonly used during inferencing operations for deployed neural networks.

[0113] **FIG. 8** illustrates a multi-GPU computing system 800, according to an embodiment. The multi-GPU computing

system 800 can include a processor 802 coupled to multiple GPGPUs 806A-806D via a host interface switch 804. The host interface switch 804, in one embodiment, is a PCI express switch device that couples the processor 802 to a PCI express bus over which the processor 802 can communicate with the set of GPGPUs 806A-806D. Each of the multiple GPGPUs 806A-806D can be an instance of the GPGPU 700 of FIG. 7. The GPGPUs 806A-806D can interconnect via a set of high-speed point-to-point GPU to GPU links 816. The high-speed GPU to GPU links can connect to each of the GPGPUs 806A-806D via a dedicated GPU link, such as the GPU link 710 as in FIG. 7. The P2P GPU links 816 enable direct communication between each of the GPGPUs 806A-806D without requiring communication over the host interface bus to which the processor 802 is connected. With GPU-to-GPU traffic directed to the P2P GPU links, the host interface bus remains available for system memory access or to communicate with other instances of the multi-GPU computing system 800, for example, via one or more network devices. While in the illustrated embodiment the GPGPUs 806A-806D connect to the processor 802 via the host interface switch 804, in one embodiment the processor 802 includes direct support for the P2P GPU links 816 and can connect directly to the GPGPUs 806A-806D.

Machine Learning Neural Network Implementations

[0114] The computing architecture provided by embodiments described herein can be configured to perform the types of parallel processing that is particularly suited for training and deploying neural networks for machine learning. A neural network can be generalized as a network of functions having a graph relationship. As is well-known in the art, there are a variety of types of neural network implementations used in machine learning. One exemplary type of neural network is the feedforward network, as previously described.

[0115] A second exemplary type of neural network is the Convolutional Neural Network (CNN). A CNN is a specialized feedforward neural network for processing data having a known, grid-like topology, such as image data. Accordingly, CNNs are commonly used for compute vision and image recognition applications, but they also may be used for other types of pattern recognition such as speech and language processing. The nodes in the CNN input layer are organized into a set of "filters" (feature detectors inspired by the receptive fields found in the retina), and the output of each set of filters is propagated to nodes in successive layers of the network. The computations for a CNN include applying the convolution mathematical operation to each filter to produce the output of that filter. Convolution is a specialized kind of mathematical operation performed by two functions to produce a third function that is a modified version of one of the two original functions. In convolutional network terminology, the first function to the convolution can be referred to as the input, while the second function can be referred to as the convolution kernel. The output may be referred to as the feature map. For example, the input to a convolution layer can be a multidimensional array of data that defines the various color components of an input image. The convolution kernel can be a multidimensional array of parameters, where the parameters are adapted by the training process for the neural network.

[0116] Recurrent neural networks (RNNs) are a family of feedforward neural networks that include feedback connections between layers. RNNs enable modeling of sequential data by sharing parameter data across different parts of the neural network. The architecture for a RNN includes cycles. The cycles represent the influence of a present value of a variable on its own value at a future time, as at least a portion of the output data from the RNN is used as feedback for processing subsequent input in a sequence. This feature makes RNNs particularly useful for language processing due to the variable nature in which language data can be composed.

[0117] The figures described below present exemplary feedforward, CNN, and RNN networks, as well as describe a general process for respectively training and deploying each of those types of networks. It will be understood that these descriptions are exemplary and nonlimiting as to any specific embodiment described herein and the concepts illustrated can be applied generally to deep neural networks and machine learning techniques in general.

[0118] The exemplary neural networks described above can be used to perform deep learning. Deep learning is machine learning using deep neural networks. The deep neural networks used in deep learning are artificial neural networks composed of multiple hidden layers, as opposed to shallow neural networks that include only a single hidden layer. Deeper neural networks are generally more computationally intensive to train. However, the additional hidden layers of the network enable multistep pattern recognition that results in reduced output error relative to shallow machine learning techniques.

[0119] Deep neural networks used in deep learning typically include a front-end network to perform feature recognition coupled to a back-end network which represents a mathematical model that can perform operations (e.g., object classification, speech recognition, etc.) based on the feature representation provided to the model. Deep learning enables machine learning to be performed without requiring hand crafted feature engineering to be performed for the model. Instead, deep neural networks can learn features based on statistical structure or correlation within the input data. The learned features can be provided to a mathematical model that can map detected features to an output. The mathematical model used by the network is generally specialized for the specific task to be performed, and different models will be used to perform different task.

[0120] Once the neural network is structured, a learning model can be applied to the network to train the network to

perform specific tasks. The learning model describes how to adjust the weights within the model to reduce the output error of the network. Backpropagation of errors is a common method used to train neural networks. An input vector is presented to the network for processing. The output of the network is compared to the desired output using a loss function and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards until each neuron has an associated error value which roughly represents its contribution to the original output. The network can then learn from those errors using an algorithm, such as the stochastic gradient descent algorithm, to update the weights of the of the neural network.

[0121] **FIG. 9A-9B** illustrate an exemplary convolutional neural network. FIG. 9A illustrates various layers within a CNN. As shown in FIG. 9A, an exemplary CNN used to model image processing can receive input 902 describing the red, green, and blue (RGB) components of an input image. The input 902 can be processed by multiple convolutional layers (e.g., convolutional layer 904, convolutional layer 906). The output from the multiple convolutional layers may optionally be processed by a set of fully connected layers 908. Neurons in a fully connected layer have full connections to all activations in the previous layer, as previously described for a feedforward network. The output from the fully connected layers 908 can be used to generate an output result from the network. The activations within the fully connected layers 908 can be computed using matrix multiplication instead of convolution. Not all CNN implementations are make use of fully connected layers 908. For example, in some implementations the convolutional layer 906 can generate output for the CNN

[0122] The convolutional layers are sparsely connected, which differs from traditional neural network configuration found in the fully connected layers 908. Traditional neural network layers are fully connected, such that every output unit interacts with every input unit. However, the convolutional layers are sparsely connected because the output of the convolution of a field is input (instead of the respective state value of each of the nodes in the field) to the nodes of the subsequent layer, as illustrated. The kernels associated with the convolutional layers perform convolution operations, the output of which is sent to the next layer. The dimensionality reduction performed within the convolutional layers is one aspect that enables the CNN to scale to process large images.

[0123] FIG. 9B illustrates exemplary computation stages within a convolutional layer of a CNN. Input to a convolutional layer 912 of a CNN can be processed in three stages of a convolutional layer 914. The three stages can include a convolution stage 916, a detector stage 918, and a pooling stage 920. The convolution layer 914 can then output data to a successive convolutional layer. The final convolutional layer of the network can generate output feature map data or provide input to a fully connected layer, for example, to generate a classification value for the input to the CNN.

[0124] In the convolution stage 916 performs several convolutions in parallel to produce a set of linear activations. The convolution stage 916 can include an affine transformation, which is any transformation that can be specified as a linear transformation plus a translation. Affine transformations include rotations, translations, scaling, and combinations of these transformations. The convolution stage computes the output of functions (e.g., neurons) that are connected to specific regions in the input, which can be determined as the local region associated with the neuron. The neurons compute a dot product between the weights of the neurons and the region in the local input to which the neurons are connected. The output from the convolution stage 916 defines a set of linear activations that are processed by successive stages of the convolutional layer 914.

[0125] The linear activations can be processed by a detector stage 918. In the detector stage 918, each linear activation is processed by a non-linear activation function. The non-linear activation function increases the nonlinear properties of the overall network without affecting the receptive fields of the convolution layer. Several types of non-linear activation functions may be used. One particular type is the rectified linear unit (ReLU), which uses an activation function defined as $f(x) = \max(0,x)$, such that the activation is thresholded at zero.

[0126] The pooling stage 920 uses a pooling function that replaces the output of the convolutional layer 906 with a summary statistic of the nearby outputs. The pooling function can be used to introduce translation invariance into the neural network, such that small translations to the input do not change the pooled outputs. Invariance to local translation can be useful in scenarios where the presence of a feature in the input data is more important than the precise location of the feature. Various types of pooling functions can be used during the pooling stage 920, including max pooling, average pooling, and l2-norm pooling. Additionally, some CNN implementations do not include a pooling stage. Instead, such implementations substitute and additional convolution stage having an increased stride relative to previous convolution stages.

[0127] The output from the convolutional layer 914 can then be processed by the next layer 922. The next layer 922 can be an additional convolutional layer or one of the fully connected layers 908. For example, the first convolutional layer 904 of FIG. 9A can output to the second convolutional layer 906, while the second convolutional layer can output to a first layer of the fully connected layers 908.

[0128] **FIG. 10** illustrates an exemplary recurrent neural network 1000. In a recurrent neural network (RNN), the previous state of the network influences the output of the current state of the network. RNNs can be built in a variety of ways using a variety of functions. The use of RNNs generally revolves around using mathematical models to predict the future based on a prior sequence of inputs. For example, an RNN may be used to perform statistical language modeling

to predict an upcoming word given a previous sequence of words. The illustrated RNN 1000 can be described has having an input layer 1002 that receives an input vector, hidden layers 1004 to implement a recurrent function, a feedback mechanism 1005 to enable a 'memory' of previous states, and an output layer 1006 to output a result. The RNN 1000 operates based on time-steps. The state of the RNN at a given time step is influenced based on the previous time step via the feedback mechanism 1005. For a given time step, the state of the hidden layers 1004 is defined by the previous state and the input at the current time step. An initial input (x_1) at a first time step can be processed by the hidden layer 1004. A second input (x_2) can be processed by the hidden layer 1004 using state information that is determined during the processing of the initial input (x_1). A given state can be computed as $s_t = f(Ux_t + Ws_{t-1})$, where U and W are parameter matrices. The function f is generally a nonlinearity, such as the hyperbolic tangent function (Tanh) or a variant of the rectifier function $f(x) = \max(0,x)$. However, the specific mathematical function used in the hidden layers 1004 can vary depending on the specific implementation details of the RNN 1000.

[0129] In addition to the basic CNN and RNN networks described, variations on those networks may be enabled. One example RNN variant is the long short term memory (LSTM) RNN. LSTM RNNs are capable of learning long-term dependencies that may be necessary for processing longer sequences of language. A variant on the CNN is a convolutional deep belief network, which has a structure similar to a CNN and is trained in a manner similar to a deep belief network. A deep belief network (DBN) is a generative neural network that is composed of multiple layers of stochastic (random) variables. DBNs can be trained layer-by-layer using greedy unsupervised learning. The learned weights of the DBN can then be used to provide pre-train neural networks by determining an optimal initial set of weights for the neural network.

[0130] **FIG. 11** illustrates training and deployment of a deep neural network. Once a given network has been structured for a task the neural network is trained using a training dataset 1102. Various training frameworks 1104 have been developed to enable hardware acceleration of the training process. For example, the machine learning framework 604 of FIG. 6 may be configured as a training framework 604. The training framework 604 can hook into an untrained neural network 1106 and enable the untrained neural net to be trained using the parallel processing resources described herein to generate a trained neural net 1108.

[0131] To start the training process the initial weights may be chosen randomly or by pre-training using a deep belief network. The training cycle then be performed in either a supervised or unsupervised manner.

[0132] Supervised learning is a learning method in which training is performed as a mediated operation, such as when the training dataset 1102 includes input paired with the desired output for the input, or where the training dataset includes input having known output and the output of the neural network is manually graded. The network processes the inputs and compares the resulting outputs against a set of expected or desired outputs. Errors are then propagated back through the system. The training framework 1104 can adjust to adjust the weights that control the untrained neural network 1106. The training framework 1104 can provide tools to monitor how well the untrained neural network 1106 is converging towards a model suitable to generating correct answers based on known input data. The training process occurs repeatedly as the weights of the network are adjusted to refine the output generated by the neural network. The training process can continue until the neural network reaches a statistically desired accuracy associated with a trained neural net 1108. The trained neural network 1108 can then be deployed to implement any number of machine learning operations.

[0133] Unsupervised learning is a learning method in which the network attempts to train itself using unlabeled data. Thus, for unsupervised learning the training dataset 1102 will include input data without any associated output data. The untrained neural network 1106 can learn groupings within the unlabeled input and can determine how individual inputs are related to the overall dataset. Unsupervised training can be used to generate a self-organizing map, which is a type of trained neural network 1107 capable of performing operations useful in reducing the dimensionality of data. Unsupervised training can also be used to perform anomaly detection, which allows the identification of data points in an input dataset that deviate from the normal patterns of the data.

[0134] Variations on supervised and unsupervised training may also be employed. Semi-supervised learning is a technique in which in the training dataset 1102 includes a mix of labeled and unlabeled data of the same distribution. Incremental learning is a variant of supervised learning in which input data is continuously used to further train the model. Incremental learning enables the trained neural network 1108 to adapt to the new data 1112 without forgetting the knowledge instilled within the network during initial training.

[0135] Whether supervised or unsupervised, the training process for particularly deep neural networks may be too computationally intensive for a single compute node. Instead of using a single compute node, a distributed network of computational nodes can be used to accelerate the training process.

[0136] **FIG. 12** is a block diagram illustrating distributed learning. Distributed learning is a training model that uses multiple distributed computing nodes to perform supervised or unsupervised training of a neural network. The distributed computational nodes can each include one or more host processors and one or more of the general-purpose processing nodes, such as the highly-parallel general-purpose graphics processing unit 700 as in FIG. 700. As illustrated, distributed learning can be performed model parallelism 1202, data parallelism 1204, or a combination of model and data parallelism

1204.

[0137] In model parallelism 1202, different computational nodes in a distributed system can perform training computations for different parts of a single network. For example, each layer of a neural network can be trained by a different processing node of the distributed system. The benefits of model parallelism include the ability to scale to particularly large models. Splitting the computations associated with different layers of the neural network enables the training of very large neural networks in which the weights of all layers would not fit into the memory of a single computational node. In some instances, model parallelism can be particularly useful in performing unsupervised training of large neural networks.

[0138] In data parallelism 1204, the different nodes of the distributed network have a complete instance of the model and each node receives a different portion of the data. The results from the different nodes are then combined. While different approaches to data parallelism are possible, data parallel training approaches all require a technique of combining results and synchronizing the model parameters between each node. Exemplary approaches to combining data include parameter averaging and update based data parallelism. Parameter averaging trains each node on a subset of the training data and sets the global parameters (e.g., weights, biases) to the average of the parameters from each node. Parameter averaging uses a central parameter server that maintains the parameter data. Update based data parallelism is similar to parameter averaging except
that instead of transferring parameters from the nodes to the parameter server, the updates to the model are transferred. Additionally, update based data parallelism can be performed in a decentralized manner, where the updates are compressed and transferred between nodes.

[0139] Combined model and data parallelism 1206 can be implemented, for example, in a distributed system in which each computational node includes multiple GPUs. Each node can have a complete instance of the model with separate GPUs within each node are used to train different portions of the model.

[0140] Distributed training has increased overhead relative to training on a single machine. However, the parallel processors and GPGPUs described herein can each implement various techniques to reduce the overhead of distributed training, including techniques to enable high bandwidth GPU-to-GPU data transfer and accelerated remote data synchronization.

Exemplary Machine Learning Applications

[0141] Machine learning can be applied to solve a variety of technological problems, including but not limited to computer vision, autonomous driving and navigation, speech recognition, and language processing. Computer vision has traditionally been one of the most active research areas for machine learning applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating new categories of visual abilities. For example, computer vision applications can be configured to recognize sound waves from the vibrations induced in objects visible in a video. Parallel processor accelerated machine learning enables computer vision applications to be trained using significantly larger training dataset than previously feasible and enables inferencing systems to be deployed using low power parallel processors.

[0142] Parallel processor accelerated machine learning has autonomous driving applications including lane and road sign recognition, obstacle avoidance, navigation, and driving control. Accelerated machine learning techniques can be used to train driving models based on datasets that define the appropriate responses to specific training input. The parallel processors described herein can enable rapid training of the increasingly complex neural networks used for autonomous driving solutions and enables the deployment of low power inferencing processors in a mobile platform suitable for integration into autonomous vehicles.

[0143] Parallel processor accelerated deep neural networks have enabled machine learning approaches to automatic speech recognition (ASR). ASR includes the creation of a function that computes the most probable linguistic sequence given an input acoustic sequence. Accelerated machine learning using deep neural networks have enabled the replacement of the hidden Markov models (HMMs) and Gaussian mixture models (GMMs) previously used for ASR.

[0144] Parallel processor accelerated machine learning can also be used to accelerate natural language processing. Automatic learning procedures can make use of statistical inference algorithms to produce models that are robust to erroneous or unfamiliar input. Exemplary natural language processor applications include automatic machine translation between human languages.

[0145] The parallel processing platforms used for machine learning can be divided into training platforms and deployment platforms. Training platforms are generally highly parallel and include optimizations to accelerate multi-GPU single node training and multi-node, multi-GPU training. Exemplary parallel processors suited for training include the highly-parallel general-purpose graphics processing unit 700 of FIG. 700 and the multi-GPU computing system 800 of FIG. 800. On the contrary, deployed machine learning platforms generally include lower power parallel processors suitable for use in products such as cameras, autonomous robots, and autonomous vehicles.

[0146] **FIG. 13** illustrates an exemplary inferencing system on a chip (SOC) 1300 suitable for performing inferencing

using a trained model. The SOC 1300 can integrate processing components including a media processor 1302, a vision processor 1304, a GPGPU 1306 and a multi-core processor 1308. The SOC 1300 can additionally include on-chip memory 1305 that can enable a shared on-chip data pool that is accessible by each of the processing components. The processing components can be optimized for low power operation to enable deployment to a variety of machine learning platforms, including autonomous vehicles and autonomous robots. For example, one implementation of the SOC 1300 can be used as a portion of the main control system for an autonomous vehicle. Where the SOC 1300 is configured for use in autonomous vehicles the SOC is designed and configured for compliance with the relevant functional safety standards of the deployment jurisdiction.

[0147] During operation, the media processor 1302 and vision processor 1304 can work in concert to accelerate computer vision operations. The media processor 1302 can enable low latency decode of multiple high-resolution (e.g., 4K, 8K) video streams. The decoded video streams can be written to a buffer in the on-chip-memory 1305. The vision processor 1304 can then parse the decoded video and perform preliminary processing operations on the frames of the decoded video in preparation of processing the frames using a trained image recognition model. For example, the vision processor 1304 can accelerate convolution operations for a CNN that is used to perform image recognition on the high-resolution video data, while back end model computations are performed by the GPGPU 1306.

[0148] The multi-core processor 1308 can include control logic to assist with sequencing and synchronization of data transfers and shared memory operations performed by the media processor 1302 and the vision processor 1304. The multi-core processor 1308 can also function as an application processor to execute software applications that can make use of the inferencing compute capability of the GPGPU 1306. For example, at least a portion of the navigation and driving logic can be implemented in software executing on the multi-core processor 1308. Such software can directly issue computational workloads to the GPGPU 1306 or the computational workloads can be issued to the multi-core processor 1308, which can offload at least a portion of those operations to the GPGPU 1306.

[0149] The GPGPU 1306 can include compute clusters such as a low power configuration of the compute clusters 706A-706H within the highly-parallel general-purpose graphics processing unit 700. The compute clusters within the GPGPU 1306 can support instruction that are specifically optimized to perform inferencing computations on a trained neural network. For example, the GPGPU 1306 can support instructions to perform low precision computations such as 8-bit and 4-bit integer vector operations.

Specialized Hardware for Efficient Machine Learning Operations

[0150] Embodiments described herein provide high-level machine learning computational primitives that can be used to abstract many of the underlying computational details of performing machine learning calculations. The high-level primitives described herein enable software logic to request high-level machine learning operations while abstracting the underlying implementation details of those operations. For example and in one embodiment, software logic can request a convolution operation for an image using a given set of filters. A single high-level instruction can be executed that has operands to define input and output buffer addresses and addresses for buffers storing filter and/or kernel data. The GPGPU can then divide the high-level convolution instruction into multiple sub-operations that are performed by the underlying compute units of the GPGPU. In one embodiment direct hardware support for one or more subroutines of the basic linear algorithm subprograms (BLAS) is provided, although embodiments can provide hardware support for other libraries of subroutines. Compiler logic and associated runtime libraries can compile source code that make use of supported high-level compute subroutines and output compiled source code that calls into a machine learning macro-instruction unit.

Instructions and Logic to Perform Compute Operations for Machine Learning

[0151] Hardware accelerators for computer vision and machine learning can improve energy efficiency for applications such as object, face and speech recognition by orders of magnitude. These accelerators use interconnected processing element (PE) arrays, with multiply-add circuits being performance, area, and energy dominant for mapping key algorithms used for CNN compute operations. For example, some machine learning hardware accelerators use narrow bit-width (16b) fixed-point multiply-add datapath building blocks to meet the stringent memory, area and power budgets for SoCs in the low-power or embedded space. Better quality of results can be achieved for some datasets and algorithms with the higher dynamic range offered by floating-point numbers/computations while still maintaining the same memory footprint (16b operands). Prior hardware solutions to accommodate both types of numeric computations employ separate fixed-point and floating-point datapath or PEs, resulting in a high area cost to achieve this flexibility. Instead, embodiment described herein provide a merged integer/floating-point fused multiply-add and multiply-accumulate datapath that utilizes existing signed integer multiply-add circuits to accomplish floating-point mantissa multiply-add operations. In one embodiment, by adding only the circuits required for alignment/normalization shifts and exponent unit, floating-point support is enabled in a combine floating-point/integer unit without increasing input/output data width and data memory footprint.

A single control signal is utilized to switch, on a per-cycle basis, between floating-point and integer compute modes.

[0152] The combined integer/floating-point units provided by embodiments are supplemented multiple types of machine learning acceleration units that can be integrated into a GPGPU. Embodiments described herein provide logic to enable additional instructions that combines a fused-multiply-add operation with a neural network activation function, such as the rectified linear unit function (RELU), the sigmoid function, or the hard-sigmoid function.

[0153] One embodiment enables an extension of the 16-bit floating-point encoding to support alternate encodings from the standard IEEE 754 half-precision floating-point format. The IEEE half-precision floating-point format specifies a 1-bit sign, a 5-bit exponent and a 10-bit fractional portion. Embodiments described herein can selectively support alternative encodings of FP16 data based on the pattern of data to be encoded. In one embodiment a supported alternative format specifies a 1-bit sign, with an 8-bit exponent and a 7-bit fractional component. One embodiment allows encoding with a 1-bit sign, a 3-bit exponent, and a 12-bit fractional component. In such embodiments, differing sets of instructions support different floating point encodings, allowing a developer to select an encoding based on the instruction specified in program code. In one embodiment, differing floating point encodings can be used when rounding or down sampling floating-point data, for example, from an accumulated 32-bit floating-point value to a 16-bit value.

[0154] The merged floating point units described herein can selectively perform 16-bit integer or floating point operations on a per-cycle basis. One embodiment enables dynamic reconfiguration of the floating-point units described herein to enable multi-format support. For example, using a multi-pass configuration, 16-bit integer or floating point units can be configured to perform a two-pass 32-bit operation or a four-pass 64-bit operation. Such logic enables floating point logic that is optimized for lower-precision inferencing operations to be clustered for use in higher-precision training operations.

[0155] One embodiment provides a stochastic rounding unit and statistic accumulator for low-precision networks. The stochastic rounding enables increased accuracy over classical quantization and rounding for low-precision deep neural networks. The rounding unit can work in different modes. A first mode is a random mode that uses a random number generator to control the rounding unit. A second mode uses a probability distribution of outputs over subsequent inputs and make use of a near-data statistics estimator unit coupled to GPGPU memory.

[0156] The techniques described herein can be implemented within a general-purpose computational system with machine learning optimizations provided via machine-learning accelerator units. A multiprocessor provided by embodiments described herein is shown in FIG. 14.

[0157] **FIG. 14** is a block diagram of a multiprocessor unit 1400, according to an embodiment. The multiprocessor unit 1400 can be a variant of a graphics multiprocessor 234 of FIG 2D. The multiprocessor unit 1400 includes a fetch and decode unit 1402, a branch unit 1404, a register file 1406, a thread manager 1406, a single instruction multiple thread unit (SIMT unit 1410), and a voltage and frequency manager 1420. The fetch and decode unit 1402 can fetch an instruction for execution by the multiprocessor unit 1400. The branch unit 1404 can compute instruction pointer adjustments based on an executed jump instruction. The register file 1406 can store general-purpose and architectural registers used by the SIMT unit 1410. The thread manager 1406 can distribute and re-distribute threads among the compute units of the SIMT unit 1410. In one embodiment, the SIMT unit 1410 is configured to execute a single instruction as multiple threads, with each thread of the instruction executed by a separate compute unit. In one embodiment compute unit 1411 through compute unit 1418 each includes an integer ALU (e.g., ALU 1411A-1418A) and a floating-point unit (e.g., FPU 1411B-1418B). The voltage and frequency of each compute unit 1411-1418 within the SIMT unit 1410 can be dynamically managed by the voltage and frequency manager 1420, which can increase or decrease the voltage and clock frequency supplied to the various compute units as components of the compute units are enabled and disabled.

[0158] In some previously enable configurations, each compute unit can execute a single thread of either an integer instruction or a floating-point instruction. If any of ALU 1411A-1418A is tasked to execute a thread of an integer instruction, respective FPU 1411B-FPU1418B is unavailable for use to execute a thread of a floating-point instruction and may be power gated during the operation of the corresponding ALU 1411A-ALU 1418A. For example, while ALU 1411A may execute a thread of an integer instruction while FPU 1413B executes a thread of a floating-point instruction, FPU 1411B is power gated while ALU 1411A is active. Embodiments described herein overcome such limitations by enabling, for example, ALU 1411A to execute thread of an instruction while FPU 1411B executes a thread of a different instruction. Furthermore, one embodiment provides support for mixed precision or mixed data-type operands, such that a single compute unit can simultaneously perform operations for an instruction having floating-point and integer operands and/or operands having different precisions.

[0159] Embodiments described herein enable increased operational throughput for a cluster of compute units by making all logic units within each compute unit available to perform computations. In such embodiments, logic units within a compute unit that are designed to perform computations selectively at one of multiple precisions or multiple data types can be configured to perform multiple simultaneous operations for each precision or data type supported by the compute unit. For a given compute unit 1411-1418, ALUs 1411A-1418A can perform integer operations while FPU 1411B-1418B perform floating-point operations. These operations can be performed for a single instruction or for multiple instruction. In one embodiment a new class of mixed-precision instruction is enabled in which one or more operands are of one data type or precision while one or more different operands are of a different data type or precision. For

example, an instruction can accept two or more multiple-element operands that include floating-point and integer data types and a single instruction is performed on a per-data type or per-precision basis.

A reconfigurable 16-bit Floating-Point/Integer Fused Multiplv-Add Unit

[0160] The logic unit designs provided by embodiments described herein have single-cycle and multi-cycle latency while maintaining single-cycle throughput for both fused multiply-add (e.g., 3-operand input with no dependence across cycles) and multiply-accumulate (e.g., 2-operand input with data dependence across cycles). In contrast, logic unit designs known in the art implement fused multiply-add without regard to multi-cycle latency and single-cycle throughput multiply-accumulate operations, which may be a limiting factor for performance for key machine-learning operations, such as the dot product operation.

[0161] One embodiment described herein provides for a merged integer/floating-point fused multiply-add datapath utilizes the existing signed integer multiply-add circuits to also accomplish floating-point mantissa multiply-add operations. With the addition of only the circuits required for alignment/normalization shifts and exponent unit, floating point support is enabled. Input/output data widths and data memory footprint remains the same, with only a single control signal required to switch on a per-cycle basis between the two compute modes.

[0162] One embodiment provides for a merged 16-bit integer/floating-point fused multiple-add design that improves upon conventional single-cycle designs with separate integer/floating-point datapaths. The designs described herein realize multiply-add circuits for a merged int16/float16 datapath that reduces total area by up to 29%. One embodiment provides for an improved floating-point datapath with alignment only for addend along with combined negation and rounding incrementer that contributes to 11% of the total area reduction. One embodiment provides for a multiply-accumulate variant with two inputs and a two-cycle latency, single-cycle throughput. One embodiment provides for an alternative circuit that significantly increases accumulation accuracy by doubling the accumulator width at a cost of only 11% in increased area.

[0163] **FIG. 15A-15B** illustrate designs for logic units to perform integer and floating-point fused multiply-add operations, according to an embodiment. **Fig. 15A** shows a conventional design for a logic unit 1500 that enables a fused multiply-add operation while maintaining full intermediate product accuracy and range. The fused multiply-add operation (o=a*b + c) is performed on three 16-bit input operands 1501 in either IEEE half-precision floating-point (float16) or signed 16b integer (int16) modes. The inputs are provided to either a 16-bit floating-point datapath 1510 or a 16-bit integer datapath 1520, with the output port (o 1530) selecting the appropriate result (f16o 1518 or i16o 1528) based on the operation mode 1532. The int16 result (i16o 1528) selects and rounds to nearest the upper half of the 32b signed integer result (isum 1525) that is generated by a signed 16bx16b multiplier 1521 and a 32b adder 1522. The float16 datapath 1510 right-shifts (1511) the mantissa of the smaller of the product of an unsigned 11bx11b multiplier 1617 and right shifts the addend for alignment at an alignment shifter 1512A before processing the product via a 22-bit mantissa adder 1513. A 22-bit leading zero anticipator (LZA 1519) predicts the location of the most significant bit location of the result of the floating-point addition performed by the 22-bit mantissa adder 1513 based on the inputs to the adder. A left-shift (1514) is performed by a normalization shifter 1515 before the intermediate result is provided to rounding logic 1516.

[0164] **FIG. 15B** is a block diagram of a multiply-add logic unit 1540, according to an embodiment. The logic unit 1540 of FIG. 15B maintains separate 16-bit floating-point/integer circuits while improving on the floating-point datapath of logic unit 1500. In one embodiment the design of logic unit 1540 removes the alignment shifter 1512B from the critical path by performing the alignment only on the addend, in parallel with the multiply operation (1541). The wider 33-bit sum only requires an 11-bit incrementer for the upper bits. Additionally, for subtract operations the output of the adder may be negated to produce an unsigned mantissa. In one embodiment the incrementer is removed from the critical path of the datapath of logic unit 1540 by combining the increment operation with the final round incrementer (1542). To the contrary, logic unit 1500 of FIG. 15A requires an incrementer to complete any required two's complement negate operations after the adder. The critical path reduction with the 16-bit floating-point datapath of logic unit 1540 results in smaller gates and allows an 11% area reduction relative to logic unit 1500 while maintaining the same single-cycle latency.

[0165] **FIG. 16** illustrates fused multiply-add logic unit 1600 having a merged floating-point and integer datapath, according to an embodiment. A 16-bit x 16-bit signed multiplier 1602A and 32-bit adder 1604 of the integer datapath are reused for floating-point mantissa operations with upper operand bits gated to produce results for 11-bit mantissas (1602B). Input switches 1601A-1601C are used to re-direct the upper 6-bits of the input operands (a, b, c) to an exponent unit 1608 when floating-point mode is enabled. Sign and exponent values from the inputs are packed and provided to an exponent unit 1608 via a fixed 3-bit sign operand bus 1609A and a 15-bit exponent bus 1609B. For 16-bit floating-point operations, the shared 32-bit adder uses a 1-bit incrementer 1605 to create the upper bit(s) 1606 of the 33-bit sum. Bypass circuits (1610A, 1610B) within the exponent unit 1608, as well as in the alignment shifter 1612 and normalization shifter 1613, ensure fixed alignment/normalization with minimal switching activity in those units for integer mode, while zero upper mantissa bits ensure no switching activity within unused portions of the multiplier in floating-point mode. The round logic 1616 and incrementer of the floating point datapath is re-used for integer mode to compute lower 10-bits of

integer result i16o with rounding. The upper 6-bits of i16o are computed by mapping that operation onto existing exponent incrementer 1611, which also performs any rounding overflow operations from the mantissa datapath in floating-point mode. A 16-bit floating point or integer value can be provided via an output 1630 when processing is complete.

[0166] **FIG. 17A** illustrates a logic unit 1700 including merged computation circuits to perform floating point and integer fused-multiply accumulate operations, according to an embodiment. The logic unit 1700 includes an exponent unit 1708 and a mantissa unit 1709, two 16-bit input ports 1701 and a 16-bit output port 1730. The input ports 1701 include switches to switch sign and exponent bits of the input data to the exponent unit 1708. The exponent unit 1708 and the mantissa unit 1709 are used when performing integer operations. In one embodiment the logic unit supports an 8.8 input and 16.0 output format for a 16-bit fixed point mode. The logic unit 1700 supports two-cycle latency and single-cycle throughput requirements. Some of the illustrated circuits are shared between operational modes, including the signed multiplier 1702A-1702B and 32-bit adder 1704, which are used for both integer and floating-point modes. A 16-bit accumulator input 1703A is asserted during accumulation in the second cycle, where the value of the accumulator is provided to the 32-bit adder 1704. The upper 10 bits of the accumulator input 1703A (e.g., c[15:6]) are exclusive to 16-bit integer operation. For both computation modes, multiplication is performed in the first cycle and addition/rounding in the second cycle.

[0167] The logic unit 1700 of FIG. 17A uses three key techniques to enable an efficient merged design. First, a straightforward pipelining of the single-cycle merged design of FIG. 16 for accumulate operations would either reduce throughput by half with addend alignment in first cycle or increases the cycle time with right-shift computation and 33b alignment in the critical path for second cycle. Instead, the design of logic unit 1700 takes advantage of the timing/area non-criticality of the exponent unit 1708 to precompute the bigger (or smaller) mantissa and right-shift amount for the alignment shifter 1713. In one embodiment the logic unit 1700 performs two-cycle operation while maintaining single-cycle throughput by feeding the output back to the second cycle as addend input, picking a smaller mantissa for only 22-bit alignment and pre-computing the smaller mantissa/right shift amount in the first cycle using the multiplier output and the accumulator exponent previously computed by the second stage.

[0168] Second, a round to nearest operation in 16-bit integer mode takes advantage of the 8.8 fixed-point format and eliminates the need to map integer rounding onto the floating-point round incrementer. A one is inserted in place of a zero at bit position 15 by the multiplexer logic 1705 before the adder to achieve the same rounding operation.

[0169] Third, flip-flops are reused for mutually exclusive signals such as exponent computation (e.g., Eun 1707, Right-shift 1710) and upper 10b of product (1711) between the two modes. Timing path reduction in the second cycle is also achieved by combining the negation /round incrementers and by using far/near path based optimizations to reduce critical path through the alignment shifter 1713 and normalization shifter 1714.

[0170] The accuracy of the two-cycle multiply-accumulate design is significantly increased by doubling the width of only the accumulator to 32-bits as shown in **FIG. 17B.** The accumulator can accumulate 16-bit integer results in a 16.16 fixed point format and 16-bit floating-point results based on an intermediate result having a 5-bit exponent and a 22-bit mantissa (implicit leading 1 not stored). The 22-bit mantissa of the intermediate result, in various embodiments, can be rounded, truncated, or quantized to an IEEE standard mantissa. The design of logic unit 1740 limits the cost of the doubled accumulator primarily to the output flip-flops and final incrementer in the mantissa datapath, as the remaining datapath after the multiplier already accommodates the additional width for the product. In one embodiment the higher accuracy enables simplifying the rounding to a simple truncation to generate a 16-bit output 1750 from the 32-bit accumulator. The post exponent normalization incrementer is removed from the exponent unit 1708 in logic unit 1740. Instead, a negation incrementer 1742 performs a final increment in the mantissa to compute the two's complement when the output of the adder is to be negated. A 32-bit accumulator input 1703B is asserted during accumulation in the second cycle, where the value of the accumulator is provided to the 32-bit adder 1704. The upper 10 bits of the accumulator input 1703B (e.g., c[31:22]) are exclusive to 16-bit integer operation. Synthesized total area of this design presents only an 11% area increase relative to the design of logic unit 1700 of FIG. 17A while doubling accumulator precision.

[0171] Even though the above descriptions are provided for 16-bit operands, these techniques can be easily extended to larger data widths to achieve similar goals. Additionally, while IEEE half precision output is described, the designs described herein may also be adjusted to support non-standard floating point formats. Additionally, different non-standard floating-point formats may be used for intermediate values, as described below.

[0172] Embodiments described above provide various implementations of a reconfigurable 16-bit Floating-Point/Integer Fused Multiply-Add Unit that provides multiple advantages over existing designs. The proposed design does not affect memory footprint for floating-point or integer number storage. The proposed designs only increase multiplier area without changing the remainder of the floating-point datapath. In contrast, logic designs known in the art extend the entire floating point significand/mantissa to the same width as the integer number, while additional storage area for sign and exponent is separate and exclusive only to floating-point numbers, resulting in an increase in register file size and footprint for floating-point number storage. Existing designs also increase width of the entire mantissa datapath, which can result in significant area increase. Single cycle (e.g., logic unit 1600 of FIG. 16) and multi-cycle (e.g., logic unit 1700 of FIG. 17A and logic unit 1740 of FIG. 17B) designs are provided, where the multi-cycle, after an initial latency, generate

an output each cycle. Logic unit 1740 of FIG. 17B provides a merged floating-point/integer multiply-accumulate design with a local accumulator width that is twice as wide as the input operands. This enables much higher accumulation accuracy for operations like dot-products without impacting memory storage footprint of input operands and affects a small portion of the design for only 11% total area impact. Furthermore, each logic unit maps a portion of the integer operation onto the existing exponent datapath to maximize circuit re-use when reconfiguring for integer mode. Additionally for floating-point operations with subtract operations, the logic unit 1540 of FIG. 15B and 1700 of FIG. 17A combine the 2's complement increment into the round increment for reduced delay and area.

Machine Learning Data Processing System and Acceleration Logic

[0173] One embodiment uses the multiprocessor unit 1400 of FIG. 14 and one or more floating-point/integer logic units of FIGs. 15A-17B can be used as building blocks for machine learning data processing system that includes hardware, software, and firmware that is optimized to perform the type of compute operations commonly performed when performing training or inferencing using deep neural networks. **FIG. 18A-18B** illustrate a data processing system and associated compute and logic units that to perform accelerated training and inferencing operations for machine learning, for example, via the use of deep neural networks. **FIG. 18A** illustrates an exemplary machine learning data processing system provided by embodiments described herein. **FIG. 18B** illustrates components of a machine learning accelerator, according to one embodiment.

[0174] The data processing system 1800 of FIG 18A is a heterogeneous processing system having a processor 1802, unified memory 1810, and a GPGPU 1820 including machine learning acceleration logic. The processor 1802 and the GPGPU 1820 can be any of the processors and GPGPU/parallel processors as described herein. The processor 1802 can execute instructions for a compiler 1815 stored in system memory 1812. The compiler 1815 executes on the processor 1802 to compile source code 1814A into compiled code 1814B. The compiled code 1814B can include code that may be executed by the processor 1802 and/or code that may be executed by the GPGPU 1820. During compilation, the compiler 1815 can perform operations to insert metadata, including hints as to the level of data parallelism present in the compiled code 1814B and/or hints regarding the data locality associated with threads to be dispatched based on the compiled code 1814B. The compiler 1815 can include the information necessary to perform such operations or the operations can be performed with the assistance of a runtime library 1816. The runtime library 1816 can also facilitate the compiler 1815 in the compilation of the source code 1814A and can also include instructions that are linked at runtime with the compiled code 1814B to facilitate execution of the compiled instructions on the GPGPU 1820.

[0175] The unified memory 1810 represents a unified address space that may be accessed by the processor 1802 and the GPGPU 1820. The unified memory includes system memory 1812 as well as GPGPU memory 1818. The GPGPU memory 1818 includes GPGPU local memory 1834A-1834B within the GPGPU 1820 and can also include some or all of system memory 1812. For example, compiled code 1814B stored in system memory 1812 can also be mapped into GPGPU memory 1818 for access by the GPGPU 1820.

[0176] The GPGPU 1820 includes multiple compute blocks 1824A-1824N, which can be instances of the processing cluster 214A-214N of **FIG. 2A** and can include one or more instances of the graphics multiprocessor 234 described herein. In various embodiments, the compute blocks 1824A-1824N include compute units having one or more of the logic units of **FIGs. 15B-17B.** The GPGPU 1820 also includes a set of registers 1825, cache memory 1827, and a power and performance module 1826 that can be used as shared resources for the compute blocks 1824A-1824N. In one embodiment the registers 1825 include directly and indirectly accessible registers, where the indirectly accessible registers may be optimized for use in matrix compute operations. The power and performance module 1826 can be configured to adjust power delivery and clock frequencies for the compute blocks 1824A-1824N to power gate idle components within the compute blocks 1824A-1824N under heavy workloads. The GPGPU 1820 includes GPGPU local memory 1828, which are physical memory modules that share a graphics card or multi-chip module with the GPGPU 1820.

[0177] In one embodiment the GPGPU 1820 includes hardware logic including a fetch and decode unit 1821, a scheduler controller 1822, and a machine learning accelerator 1823. The instruction fetch and decode unit 1821 is a fetch and decode unit including logic to fetch and decode instructions, including machine learning specific instructions, that can define complex, customizable behavior. The instructions can cause the compute logic to schedule, via the scheduler controller 1822, a set of operations to be performed via one or more of the compute blocks 1824A-1824N. In one embodiment the scheduler controller 1822 is an ASIC configurable to perform advanced scheduling operations. In one embodiment the scheduler controller 1822 is a micro-controller or a low energy-per-instruction processing core capable of performing instructions loaded from a firmware module.

[0178] In one embodiment some functions to be performed by the compute blocks 1824A-1824N can be directly scheduled to or offloaded to the machine learning accelerator 1823. The machine learning accelerator 1823 includes processing element logic configured to efficiently perform matrix and other compute operations on commonly performed during machine learning.

[0179] In some embodiments the GPGPU 1820 additionally incudes a statistics unit 1829 that may be configured as

a near-data compute unit. For example, the statistics unit 1829 may be integrated into or spread across one or more memory controllers for the GPGPU local memory 1828. In one embodiment the statistics unit 1829, when enabled by the machine learning accelerator 1823, can be used to determine a probability distribution for weight or activation map data when performing machine-learning operations that write to or read from the GPGPU local memory 1828. The statistics unit 1829 includes logic to determine, based on address and data patterns during memory access, whether data that is accessed in the GPGPU local memory 1828 fits within one or more statistical distributions (e.g., Gaussian, uniform, Poisson etc.). In one embodiment, statistical information (e.g., mean, median, mode, standard deviation, etc.) can be gathered during sample period for at least a subset of memory accesses. The statistics unit 1829 can be configured such that gathering the statistical information does not significantly increase the latency of memory accesses performed through a memory controller hosting the statistics unit 1829. The statistical information can be periodically provided to the machine learning accelerator 1823 or the machine learning accelerator 1823 can request the data from the statistics unit. In one embodiment the statistics unit 1829 can check data associated with memory accesses against a set of known-likely distributions. A vector including a set of probabilities associated with each of the known-likely distributions can be provided to the machine learning accelerator 1823 on a periodic basis or upon request. In various embodiments the machine learning accelerator 1823 can use the probability and/or statistical information provided by the statistics unit 1829 for a variety of operation. In one embodiment, as described further in FIG. 18B and FIG. 20, the machine learning accelerator 1823 can use data provided by the statistics unit 1829 to perform stochastic rounding during quantization for low precision neural networks.

[0180] The machine learning accelerator 1823 of FIG. 18A is illustrated in further detail in FIG. 18B. In one embodiment the machine learning accelerator 1823 includes an activation instruction module 1832, an FPU encoding and configuration module 1834, a stochastic quantization unit 1838, and a cache memory 1836 that is shared between the various modules within the machine learning accelerator 1823.

[0181] The activation instruction module 1832 includes logic to sequence the performance of a combined fused multiply-add and activation in response to a single instruction. In response to decode of a FMAC or FMADD plus activation function on the GPGPU 1820, the scheduler unit 1822 can schedule an operation via the machine learning accelerator 1823. The machine learning accelerator 1823, via the activation instruction module 1832, can perform a set of fused multiply-add or fused multiply-accumulate operations on two or three input operands per thread or vector element and, for each thread or element, provide the output to hardware logic configure to perform one of multiple selectable activation functions. A different activation function can be associated with different instructions or a single instruction can include a field to enable selection of an activation function. In one embodiment, the activation instruction module can perform a vector or warp operation to generate intermediate FMADD or FMAC result and store the intermediate results in the cache memory 1836. The activation instruction module 1832 can then apply the activation function to the intermediate data. Exemplary supported activation functions include the rectified linear unit (RELU) function of equation (1), the sigmoid function of equation (2), or the hard-sigmoid function of equation (3).

$$f(x) = \max(0, x) \qquad (1)$$

$$\sigma(x) = \frac{1}{(1+e^{-x})} \qquad (2)$$

$$\sigma(x) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \qquad (3)$$

[0182] The FPU encoding and configuration module 1834 includes logic to define parameters for the dynamic configuration of floating point units within the compute blocks 1824A-1824N of the GPGPU 1820. In one embodiment certain dynamic aspects of the merged integer/floating-point units of **FIG. 16** and **FIGs. 17A-17B** can be configured via the FPU encoding and configuration module 1834. For example, the compute blocks 1825A-1824N can be overprovisioned to contain more compute units than can be maximally active at any one time given the power budget of the GPGPU 1820. However, the FPU encoding and configuration module 1834 can configure the dynamic floating point units to gate certain logic blocks to operate at reduce precision and reduced power draw. The reduced precision and power requirements of each unit can enable a larger number of units to be online, allowing a larger number of threads to be performed for lower-precision operations. For example and in one embodiment, logic units that may be configure to perform 16-bit integer operations can be configured to perform 8-bit integer operations, reducing power requirements. In one embodiment, dual 8-bit integer operations can be performed, increasing throughout without significantly increasing power draw. In one embodiment, multiple half-precision logic units can work in parallel to perform single precision or double precision

floating point operations. In one embodiment, higher precision operations can be performed via multiple passes through the logic unit.

[0183] In one embodiment the FPU encoding and configuration module 1834 can also configure the floating-point encoding methods supported by the floating point units. In addition to IEEE 754 floating-point standards for half, single, and double precision encoding for floating point values, a myriad of alternative encoding formats may be supported based on the dynamic range of the data that is currently being processed. For example, based on the dynamic range and/or distribution a given dataset, the data may be quantized more accurately from higher to lower precision by using greater than or fewer bits for exponent or mantissa data. In one embodiment a supported alternative format specifies a 1-bit sign, with an 8-bit exponent and a 7-bit fractional component. One embodiment allows encoding with a 1-bit sign, a 3-bit exponent, and a 12-bit fractional component. In such embodiments, differing sets of instructions support different floating point encodings, allowing a developer to select an encoding based on the instruction specified in program code. In one embodiment, differing floating point encodings can be used when rounding or down sampling floating-point data, for example, from an accumulated 32-bit floating-point value to a 16-bit value. In one embodiment the statistics unit 1829 can be leveraged to determine which 16-bit encoding is best suited for a given block of data.

[0184] In one embodiment the machine learning accelerator 1823 additionally includes stochastic quantization unit 1838 to enable stochastic quantization for machine learning operations. The stochastic quantization unit 1838 can be used to enable stochastic rounding during quantization operations. One embodiment enables stochastic rounding using a random number generator, where a fractional value can be used to determine a rounding probability. One embodiment makes use of the statistics unit 1829 to determine a probability distribution associated with the set of output data from a given layer of a neural network. For each layer, a probability density of the data values can be determined, where the probability density is determined by statistical characteristics including the mean, standard deviation, and variance of the data determined for each layer of the neural network. Using such data, stochastic rounding can be performed in a manner that does not alter the probability distribution of the data within each layer of the neural network.

[0185] **FIG. 19** illustrates details of the activation instruction module 1832, according to an embodiment. The activation instruction module 1832 includes logic to sequence the performance of a combined fused multiply-add and activation in response to a single instruction. In response to decode of a FMAC/FMADD + Activation function by the instruction fetch and decode unit 1821 of FIG. 18A, instruction execution can be dispatched to the activation instruction module 1832 via the machine learning accelerator 1823. The machine learning accelerator 1823, upon receipt of the instruction, can use a fused multiply-add/fused multiply-accumulate thread scheduler unit 1902 to schedule a set of fused multiply-add or fused multiply-accumulate operations to compute units within compute blocks 1824A-1824N. In one embodiment, intermediate data output from the compute blocks 1824A-1824N can be stored in cache memory 1836 within the machine learning accelerator 1823. In one embodiment, chunks of intermediate data can be processed in a streaming manner within the activation instruction module 1832. The intermediate data, in one embodiment, can represent an activation map to which the non-linearity of the activation function will be applied. A selected one of the activation functions can be applied by activation function logic 1904A-1904N. The activation function can be selected based on the specific instruction processed by the activation instruction module 1832 or parameters supplied with the instruction. The specific instruction can be formatted based on any of the instruction formats described herein.

[0186] Floating-point operations, at various points, include a rounding operation. Rounding is used in floating point calculations because floating point numbers have a limited number of digits and cannot represent all real numbers accurately. Thus, when a number is tasked to represent a value that requires more digits than the selected floating point format allows, the leftover digits are omitted and the number is rounded to the nearest value that may be represented by the floating-point format. The specific numbers that may be represented is dependent upon the floating-point format selected.

[0187] Various approaches to rounding during floating point calculations may be performed. Embodiments described herein include hardware logic to perform stochastic rounding for machine learning operations. In contrast with other rounding approaches that round to the nearest number or strictly up and down, the stochastic approach rounds numbers randomly. Embodiments described herein enable stochastic rounding for quantization of data values for deep neural networks. A rounding unit is provided that enables hardware stochastic rounding using one of multiple rounding modes. One embodiment enables stochastic rounding using a random number generator. A fractional value can be used to determine a rounding probability. The random number can be compared with the rounding probability to determine which of the nearest representable value to round during quantization. Alternatively, one embodiment makes use of statistics accumulator/estimator logic to determine a probability distribution associated with the set of output data from a given layer of a neural network. For each layer, a probability density of the distribution of the data values can be determined, where the probability density is defined by the mean, standard deviation, and variance of the data determined for each layer of the neural network. Using such data, stochastic rounding can be performed in a manner that does not alter the probability distribution for each layer of the neural network.

[0188] **FIG. 20** illustrates the stochastic quantization unit 1838, according to an embodiment. In one embodiment the stochastic quantization unit 1838 is used to quantize raw output data generated within a layer of a neural network into

the format used by the next layer of the neural network. For example, the computation operations used to generate output data may be processed at higher precision and the results may be quantized to a lower precision before being provided as input to the next layer. In one embodiment, the output 2002B from a given layer n is processed, for example, at 32-bits and quantized by the quantization unit 2004 into a 16-bit data type. The quantization operation can make use of stochastic rounding, which may be implemented via a stochastic rounding unit 2009. The quantized and rounded values can then be provided to the next layer (Layer N+1) 2010 of the neural network.

[0189] In various embodiments the stochastic quantization unit 1838 can perform stochastic rounding via the use of a random number generator 2006. In floating-point arithmetic, rounding aims to turn a given value *x* into a value *z* with a specified number of significant digits, where, *z* is a multiple of a number *m* that depends on the magnitude of *x*. The number *m* is a power of the base (usually 2 or 10) of the floating-point representation. The number *z* is a representable value that is proximate to the value *x*. Whether the value *x* is rounded up or down to realize the value *z* is based on a random value that is selected by the random number generator 2006. The random value that is generated is compared with the fractional portion between valid representations. The fractional portion can be used as the probability of rounding up or down to the nearest representable value. The gap between the representable values during quantization depends on the encoding format for the floating-point representation in place. As an example, if the quantization is to round to an integer value and the fractional value is 0.3, the probability of rounding up can be equated to 30%, while the probability of rounding down can be equated to 70%. In such scenario, where the random number generator 2006 is a properly validated, true random number generator, the stochastic rounding unit 2009 will round up or down in proportion to the fractional value.

[0190] Alternatively, the stochastic rounding unit 2009 can make use of a statistics accumulator/estimator 2008, which, in one embodiment, is a near-data statistics unit 1829 as in FIG. 18A. The statistics accumulator/estimator 2008 can analyze output from previous layers 2002A-2002B to determine the distribution associated with the neural network data. The stochastic rounding unit 2009 can then round data during quantization such that the quantized data has a similar distribution as the pre-quantized data.

[0191] **FIG. 21** illustrates the FPU encoding and configuration module 1834, according to one embodiment. In one embodiment the GPU encoding and configuration module 1834 includes an FPU configuration module 2102 and an FPU encoding module 2104. The FPU configuration module 2102 can be used to configure to perform 16-bit integer logic units to perform 8-bit integer operations, including dual 8-bit integer operations. In one embodiment multiple half-precision logic units can work in parallel to perform single precision or double precision floating point operations. The FPU encoding module 2104 can be used to configure the specific floating-point encoding format to use within the computation blocks 1824A-1824N during data computations. In one embodiment the FPU encoding module 2104 can configure one or more of the compute blocks 1824A-1824N in response to an instruction that specifies that input or output data is to be stored in a non-standard floating-point format. The compute blocks to execute the instruction can then be configured to interpret data in the non-standard format before operations of the instructions are executed. In one embodiment the FPU encoding module 2104 is to configure one or more of the compute blocks to use a floating-point encoding format that can most efficiently store the data to be processed. Such determination can be performed in part based on probability and statistics information provided by the statistics unit 1829, which can function as a near-data compute unit that is situated within a memory controller 2106 of the GPGPU local memory 1828.

[0192] **FIG. 22** illustrates logic 2200 to process an instruction using a dynamically configurable compute unit, according to an embodiment. The logic 2200 can be hardware or firmware logic within a GPGPU and/or GPGPU multiprocessor as described herein, such as the multiprocessor unit 1400 as in FIG. 14, or the GPGPU 1820 of FIG. 18. The logic 2200 is configured to fetch and decode a single instruction to perform a combined multiply-add operation on a set of operands, as shown at block 2202. The logic 2200 can then issue the single instruction for execution by a compute unit for execution by dynamically configurable compute unit, as shown at block 2204. The logic 2200 can then configure one or more logic units of the compute unit to perform operations at the precision and data type of the operands, as shown at block 2206. The logic 2200 can then execute the single instruction at the compute unit to generate an output based on a multiply and add operation, as shown at block 2208.

[0193] In one embodiment the combined multiply and add operation performed at block 2202 can be a fused floating-point operation including a single rounding. For example, the multiply and add operation can be a fused multiply-add or a fused multiply-accumulate operation. The combined multiply and add operation can also be an integer operation. The integer operation can include a round operation between the multiply and the add. The round can be performed by inserting a zero at the highest bit position of the integer data type via a multiplexer within the logic unit. The multiplexer can be positioned after the multiplier and before the adder within the logic unit.

[0194] In one embodiment the dynamically configurable logic unit of block 2204 is a merged floating-point and integer logic unit that is configurable to perform integer or floating-point operations. For example, the dynamically configurable logic unit may be one of logic unit 1600 of FIG. 16, 1700 of FIG 17A, or 1740 of FIG. 17B. The compute unit can include multiple different instances of such logic units. In one embodiment the logic unit is configurable on a per-cycle basis. In one embodiment the logic unit is a first logic unit configured to perform a single-cycle fused-multiply add operation using

a multiplier and an adder that are shared between floating-point and integer datapaths. In one embodiment the logic unit is a second logic unit configured to perform a two-cycle fused multiply accumulate operation having single-cycle throughput. In one embodiment the logic unit is a third logic unit configured to perform a two-cycle fused multiply accumulate operation, where the third logic includes an accumulator having twice the bit-width of the input and output operands. In one embodiment the die area of the third logic unit is at most eleven percent greater than the die area of the second logic unit.

[0195] The dynamically configurable logic units described herein can be configured to perform integer or floating-point operations. In one embodiment one or more of the logic units can be configured to perform operations at multiple different precisions. In one embodiment the logic units can be used to perform operations at multiple different precisions via multi-cycle operations. In one embodiment, different floating-point encodings may be selected, including the IEEE 754 half-precision floating-point format, single precision floating-point format, and double precision floating-point format. Non-standard floating point formats may also be used in which different bit allocations are used for the exponent and mantissa of the floating-point values.

[0196] In one embodiment the output based on the multiply and add operation can then be additionally processed by an activation function. For example, in response to a single instruction, an FMADD or FMAC operation can be scheduled by an FMADD/FMAC thread scheduler unit, as shown in FIG. 19. The output of such operations may be activation map data that can be provided to activation function logic (e.g., activation function logic 1904 as in FIG. 19) to generate neuron activation data.

[0197] **FIG. 23A** illustrates logic 2300 to execute a machine learning instruction, according to an embodiment. The logic 2300 can be hardware or firmware logic within a GPGPU and/or GPGPU multiprocessor as described herein, such as the multiprocessor unit 1400 as in FIG. 14, or the GPGPU 1820 of FIG. 18. The logic 2300 is configured to fetch and decode a single instruction to perform a set of machine learning operations via a machine learning accelerator unit, as shown at block 2302. The machine learning accelerator unit includes element of the machine learning accelerator 1823 described herein, including the activation instruction module 1832, FPU encoding and configuration module 1834, and stochastic quantization unit 1838 of FIG. 18B. The logic 2300 can then issue the single instruction for execution by a set of dynamically configurable compute units, as shown at block 2304. The logic can then configure the set of compute units to perform the set of machine learning operations at a higher precision than the inputs and outputs of the operations, as shown at block 2306. In one embodiment the configuration is performed by an FPU configuration module as described herein. The FPU configuration module can configure the compute units to perform, for example, a convolution operation on 16-bit floating point matrix data using 32-bit intermediate data. As shown at block 2308, the logic 2300 can then quantize the higher precision intermediate values to a lower precision before output via stochastic rounding logic within the machine learning accelerator. For example, 32-bit intermediate data can be quantized to 16-bits for output using stochastic rounding.

[0198] **FIG. 23B** illustrates logic 2310 to configure floating-point operations based on a distribution of neural network data, according to an embodiment. In one embodiment the logic 2300 includes hardware and firmware logic and logic units described herein, including a stochastic quantization unit 1838 of FIG. 18B and FIG 20, the FPU encoding and configuration module 1834 of FIG 18B. The statistics accumulator/estimator 2008 of FIG. 20, in one embodiment, is included within the statistics unit 1829 of FIG. 18A. The statistics unit 1829 may be a near-data compute unit included within a memory controller for the GPGPU, as shown in FIG. 21.

[0199] Using the statistics unit, the logic 2310 can determine a set of statistical metrics for neural network data stored in memory, as shown at block 2312. The logic 2310 can then determine, via the statistical metrics, a distribution for the neural network data in memory, as shown at block 2314. In one embodiment the logic 2310 can configure floating-point encoding for the compute units for use in performing a set of machine learning operations, as shown at block 2316. The logic 2310 can then configure stochastic rounding logic within the machine learning accelerator to round based on the distribution, as shown at block 2318. The stochastic rounding logic can be configured to round based on the distribution such that the probability distribution of the quantized neural network data is closer to the pre-quantized data than may be possible using random number generator based stochastic rounding techniques.

Additional Exemplary Graphics Processing System

[0200] Details of the embodiments described above can be incorporated within graphics processing systems and devices described below. The graphics processing system and devices of **FIG. 24** through **FIG. 37** illustrate alternative systems and graphics processing hardware that can implement any and all of the techniques described above.

Additional Exemplary Graphics Processing System Overview

[0201] **FIG. 24** is a block diagram of a processing system 2400, according to an embodiment. In various embodiments the system 2400 includes one or more processors 2402 and one or more graphics processors 2408, and may be a single

processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors 2402 or processor cores 2407. In one embodiment, the system 2400 is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

[0202] An embodiment of system 2400 can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments system 2400 is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system 2400 can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system 2400 is a television or set top box device having one or more processors 2402 and a graphical interface generated by one or more graphics processors 2408.

[0203] In some embodiments, the one or more processors 2402 each include one or more processor cores 2407 to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores 2407 is configured to process a specific instruction set 2409. In some embodiments, instruction set 2409 may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores 2407 may each process a different instruction set 2409, which may include instructions to facilitate the emulation of other instruction sets. Processor core 2407 may also include other processing devices, such a Digital Signal Processor (DSP).

[0204] In some embodiments, the processor 2402 includes cache memory 2404. Depending on the architecture, the processor 2402 can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor 2402. In some embodiments, the processor 2402 also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores 2407 using known cache coherency techniques. A register file 2406 is additionally included in processor 2402 which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor 2402.

[0205] In some embodiments, processor 2402 is coupled with a processor bus 2410 to transmit communication signals such as address, data, or control signals between processor 2402 and other components in system 2400. In one embodiment the system 2400 uses an exemplary 'hub' system architecture, including a memory controller hub 2416 and an Input Output (I/O) controller hub 2430. A memory controller hub 2416 facilitates communication between a memory device and other components of system 2400, while an I/O Controller Hub (ICH) 2430 provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub 2416 is integrated within the processor.

[0206] Memory device 2420 can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device 2420 can operate as system memory for the system 2400, to store data 2422 and instructions 2421 for use when the one or more processors 2402 executes an application or process. Memory controller hub 2416 also couples with an optional external graphics processor 2412, which may communicate with the one or more graphics processors 2408 in processors 2402 to perform graphics and media operations.

[0207] In some embodiments, ICH 2430 enables peripherals to connect to memory device 2420 and processor 2402 via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller 2446, a firmware interface 2428, a wireless transceiver 2426 (e.g., Wi-Fi, Bluetooth), a data storage device 2424 (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller 2440 for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers 2442 connect input devices, such as keyboard and mouse 2444 combinations. A network controller 2434 may also couple with ICH 2430. In some embodiments, a high-performance network controller (not shown) couples with processor bus 2410. It will be appreciated that the system 2400 shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub 2430 may be integrated within the one or more processor 2402, or the memory controller hub 2416 and I/O controller hub 2430 may be integrated into a discreet external graphics processor, such as the external graphics processor 2412.

[0208] **FIG. 25** is a block diagram of an embodiment of a processor 2500 having one or more processor cores 2502A-2502N, an integrated memory controller 2514, and an integrated graphics processor 2508. Those elements of **FIG. 25** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor 2500 can include additional cores up to and including additional core 2502N represented by the dashed lined boxes. Each of processor cores 2502A-2502N includes one or more internal cache units 2504A-2504N. In some embodiments each processor core also has access to one or more shared cached units 2506.

[0209] The internal cache units 2504A-2504N and shared cache units 2506 represent a cache memory hierarchy

within the processor 2500. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units 2506 and 2504A-2504N.

[0210] In some embodiments, processor 2500 may also include a set of one or more bus controller units 2516 and a system agent core 2510. The one or more bus controller units 2516 manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core 2510 provides management functionality for the various processor components. In some embodiments, system agent core 2510 includes one or more integrated memory controllers 2514 to manage access to various external memory devices (not shown).

[0211] In some embodiments, one or more of the processor cores 2502A-2502N include support for simultaneous multi-threading. In such embodiment, the system agent core 2510 includes components for coordinating and operating cores 2502A-2502N during multi-threaded processing. System agent core 2510 may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores 2502A-2502N and graphics processor 2508.

[0212] In some embodiments, processor 2500 additionally includes graphics processor 2508 to execute graphics processing operations. In some embodiments, the graphics processor 2508 couples with the set of shared cache units 2506, and the system agent core 2510, including the one or more integrated memory controllers 2514. In some embodiments, a display controller 2511 is coupled with the graphics processor 2508 to drive graphics processor output to one or more coupled displays. In some embodiments, display controller 2511 may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor 2508 or system agent core 2510.

[0213] In some embodiments, a ring based interconnect unit 2512 is used to couple the internal components of the processor 2500. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some embodiments, graphics processor 2508 couples with the ring interconnect 2512 via an I/O link 2513.

[0214] The exemplary I/O link 2513 represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module 2518, such as an eDRAM module. In some embodiments, each of the processor cores 2502A-2502N and graphics processor 2508 use embedded memory modules 2518 as a shared Last Level Cache.

[0215] In some embodiments, processor cores 2502A-2502N are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores 2502A-2502N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores 2502A-2502N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores 2502A-2502N are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor 2500 can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

[0216] **FIG. 26** is a block diagram of a graphics processor 2600, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor 2600 includes a memory interface 2614 to access memory. Memory interface 2614 can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

[0217] In some embodiments, graphics processor 2600 also includes a display controller 2602 to drive display output data to a display device 2620. Display controller 2602 includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In some embodiments, graphics processor 2600 includes a video codec engine 2606 to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

[0218] In some embodiments, graphics processor 2600 includes a block image transfer (BLIT) engine 2604 to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) 2610. In some embodiments, GPE 2610 is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

[0219] In some embodiments, GPE 310 includes a 3D pipeline 2612 for performing 3D operations, such as rendering

three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline 2612 includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system 2615. While 3D pipeline 2612 can be used to perform media operations, an embodiment of GPE 2610 also includes a media pipeline 2616 that is specifically used to perform media operations, such as video post-processing and image enhancement.

[0220] In some embodiments, media pipeline 2616 includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine 2606. In some embodiments, media pipeline 2616 additionally includes a thread spawning unit to spawn threads for execution on 3D/Media sub-system 2615. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media sub-system 2615.

[0221] In some embodiments, 3D/Media subsystem 2615 includes logic for executing threads spawned by 3D pipeline 2612 and media pipeline 2616. In one embodiment, the pipelines send thread execution requests to 3D/Media subsystem 2615, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In some embodiments, 3D/Media subsystem 2615 includes one or more internal caches for thread instructions and data. In some embodiments, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

Additional Exemplary Graphics Processing Engine

[0222] **FIG. 27** is a block diagram of a graphics processing engine 2710 of a graphics processor in accordance with some embodiments. In one embodiment, the graphics processing engine (GPE) 2710 is a version of the GPE 2610 shown in **FIG. 26.** Elements of **FIG. 27** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. For example, the 3D pipeline 2612 and media pipeline 2616 of **FIG. 26** are illustrated. The media pipeline 2616 is optional in some embodiments of the GPE 2710 and may not be explicitly included within the GPE 2710. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE 2710.

[0223] In some embodiments, GPE 2710 couples with or includes a command streamer 2703, which provides a command stream to the 3D pipeline 2612 and/or media pipelines 2616. In some embodiments, command streamer 2703 is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer 2703 receives commands from the memory and sends the commands to 3D pipeline 2612 and/or media pipeline 2616. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline 2612 and media pipeline 2616. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline 2612 can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline 2612 and/or image data and memory objects for the media pipeline 2616. The 3D pipeline 2612 and media pipeline 2616 process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to a graphics core array 2714.

[0224] In various embodiments the 3D pipeline 2612 can execute one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array 2714. The graphics core array 2714 provides a unified block of execution resources. Multi-purpose execution logic (e.g., execution units) within the graphic core array 2714 includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

[0225] In some embodiments the graphics core array 2714 also includes execution logic to perform media functions, such as video and/or image processing. In one embodiment, the execution units additionally include general-purpose logic that is programmable to perform parallel general purpose computational operations, in addition to graphics processing operations. The general purpose logic can perform processing operations in parallel or in conjunction with general purpose logic within the processor core(s) 2407 of **FIG. 24** or core 2502A-2502N as in **FIG. 25.**

[0226] Output data generated by threads executing on the graphics core array 2714 can output data to memory in a unified return buffer (URB) 2718. The URB 2718 can store data for multiple threads. In some embodiments the URB 2718 may be used to send data between different threads executing on the graphics core array 2714. In some embodiments the URB 2718 may additionally be used for synchronization between threads on the graphics core array and fixed function logic within the shared function logic 2720.

[0227] In some embodiments, graphics core array 2714 is scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE 2710. In one embodiment the execution resources are dynamically scalable, such that execution resources may

be enabled or disabled as needed.

[0228] The graphics core array 2714 couples with shared function logic 2720 that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic 2720 are hardware logic units that provide specialized supplemental functionality to the graphics core array 2714. In various embodiments, shared function logic 2720 includes but is not limited to sampler 2721, math 2722, and inter-thread communication (ITC) 2723 logic. Additionally, some embodiments implement one or more cache(s) 2725 within the shared function logic 2720. A shared function is implemented where the demand for a given specialized function is insufficient for inclusion within the graphics core array 2714. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic 2720 and shared among the execution resources within the graphics core array 2714. The precise set of functions that are shared between the graphics core array 2714 and included within the graphics core array 2714 varies between embodiments.

[0229] **FIG. 28** is a block diagram of another embodiment of a graphics processor 2800. Elements of **FIG. 28** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0230] In some embodiments, graphics processor 2800 includes a ring interconnect 2802, a pipeline front-end 2804, a media engine 2837, and graphics cores 2880A-2880N. In some embodiments, ring interconnect 2802 couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In some embodiments, the graphics processor is one of many processors integrated within a multi-core processing system.

[0231] In some embodiments, graphics processor 2800 receives batches of commands via ring interconnect 2802. The incoming commands are interpreted by a command streamer 2803 in the pipeline front-end 2804. In some embodiments, graphics processor 2800 includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) 2880A-2880N. For 3D geometry processing commands, command streamer 2803 supplies commands to geometry pipeline 2836. For at least some media processing commands, command streamer 2803 supplies the commands to a video front end 2834, which couples with a media engine 2837. In some embodiments, media engine 2837 includes a Video Quality Engine (VQE) 2830 for video and image post-processing and a multi-format encode/decode (MFX) 2833 engine to provide hardware-accelerated media data encode and decode. In some embodiments, geometry pipeline 2836 and media engine 2837 each generate execution threads for the thread execution resources provided by at least one graphics core 2880A.

[0232] In some embodiments, graphics processor 2800 includes scalable thread execution resources featuring modular cores 2880A-2880N (sometimes referred to as core slices), each having multiple sub-cores 2850A-550N, 2860A-2860N (sometimes referred to as core sub-slices). In some embodiments, graphics processor 2800 can have any number of graphics cores 2880A through 2880N. In some embodiments, graphics processor 2800 includes a graphics core 2880A having at least a first sub-core 2850A and a second sub-core 2860A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., 2850A). In some embodiments, graphics processor 2800 includes multiple graphics cores 2880A-2880N, each including a set of first sub-cores 2850A-2850N and a set of second sub-cores 2860A-2860N. Each sub-core in the set of first sub-cores 2850A-2850N includes at least a first set of execution units 2852A-2852N and media/texture samplers 2854A-2854N. Each sub-core in the set of second sub-cores 2860A-2860N includes at least a second set of execution units 2862A-2862N and samplers 2864A-2864N. In some embodiments, each sub-core 2850A-2850N, 2860A-2860N shares a set of shared resources 2870A-2870N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

Additional Exemplary Execution Units

[0233] **FIG. 29** illustrates thread execution logic 2900 including an array of processing elements employed in some embodiments of a GPE. Elements of **FIG. 29** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0234] In some embodiments, thread execution logic 2900 includes a shader processor 2902, a thread dispatcher 2904, instruction cache 2906, a scalable execution unit array including a plurality of execution units 2908A-2908N, a sampler 2910, a data cache 2912, and a data port 2914. In one embodiment the scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution unit 2908A, 2908B, 2908C, 2908D, through 2908N-1 and 2908N) based on the computational requirements of a workload. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic 2900 includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache 2906, data port 2914, sampler 2910, and execution units 2908A-2908N. In some embodiments, each execution unit (e.g. 2908A) is a stand-alone programmable general purpose com-

putational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units 2908A-2908N is scalable to include any number individual execution units.

[0235] In some embodiments, the execution units 2908A-2908N are primarily used to execute shader programs. A shader processor 2902 can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher 2904. In one embodiment the thread dispatcher includes logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution unit in the execution units 2908A-2908N. For example, the geometry pipeline (e.g., 2836 of **FIG. 28**) can dispatch vertex, tessellation, or geometry shaders to the thread execution logic 2900 (**FIG. 29**) for processing. In some embodiments, thread dispatcher 2904 can also process runtime thread spawning requests from the executing shader programs.

[0236] In some embodiments, the execution units 2908A-2908N support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units 2908A-2908N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units 2908A-2908N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader.

[0237] Each execution unit in execution units 2908A-2908N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units 2908A-2908N support integer and floating-point data types.

[0238] The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

[0239] One or more internal instruction caches (e.g., 2906) are included in the thread execution logic 2900 to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 2912) are included to cache thread data during thread execution. In some embodiments, a sampler 2910 is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler 2910 includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

[0240] During execution, the graphics and media pipelines send thread initiation requests to thread execution logic 2900 via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor 2902 is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, a pixel shader or fragment shader calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel processor logic within the shader processor 2902 then executes an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor 2902 dispatches threads to an execution unit (e.g., 2908A) via thread dispatcher 2904. In some embodiments, pixel shader 2902 uses texture sampling logic in the sampler 2910 to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

[0241] In some embodiments, the data port 2914 provides a memory access mechanism for the thread execution logic 2900 output processed data to memory for processing on a graphics processor output pipeline. In some embodiments,

the data port 2914 includes or couples to one or more cache memories (e.g., data cache 2912) to cache data for memory access via the data port.

[0242] **FIG. 30** is a block diagram illustrating a graphics processor instruction formats 3000 according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format 3000 described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

[0243] In some embodiments, the graphics processor execution units natively support instructions in a 128-bit instruction format 3010. A 64-bit compacted instruction format 3030 is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format 710 provides access to all instruction options, while some options and operations are restricted in the 64-bit format 3030. The native instructions available in the 64-bit format 3030 vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field 3013. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format 3010.

[0244] For each format, instruction opcode 3012 defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field 3014 enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For instructions in the 128-bit instruction format 3010 an exec-size field 3016 limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field 3016 is not available for use in the 64-bit compact instruction format 3030.

[0245] Some execution unit instructions have up to three operands including two source operands, src0 3020, src1 3022, and one destination 3018. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 3024), where the instruction opcode 3012 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

[0246] In some embodiments, the 128-bit instruction format 3010 includes an access/address mode field 3026 specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

[0247] In some embodiments, the 128-bit instruction format 3010 includes an access/address mode field 3026, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode is used to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

[0248] In one embodiment, the address mode portion of the access/address mode field 3026 determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

[0249] In some embodiments instructions are grouped based on opcode 3012 bit-fields to simplify Opcode decode 3040. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group 3042 includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group 3042 shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group 3044 (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group 3046 includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group 3048 includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group 3048 performs the arithmetic operations in parallel across data channels. The vector math group 3050 includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

Additional Exemplary Graphics Pipeline

[0250] **FIG. 31** is a block diagram of another embodiment of a graphics processor 3100. Elements of **FIG. 31** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

[0251] In some embodiments, graphics processor 3100 includes a graphics pipeline 3120, a media pipeline 3130, a display engine 3140, thread execution logic 3150, and a render output pipeline 3170. In some embodiments, graphics processor 3100 is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor 3100 via a ring interconnect 3102. In some embodiments, ring interconnect 3102 couples graphics processor 3100 to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect 3102 are interpreted by a command streamer 3103, which supplies instructions to individual components of graphics pipeline 3120 or media pipeline 3130.

[0252] In some embodiments, command streamer 3103 directs the operation of a vertex fetcher 3105 that reads vertex data from memory and executes vertex-processing commands provided by command streamer 3103. In some embodiments, vertex fetcher 3105 provides vertex data to a vertex shader 3107, which performs coordinate space transformation and lighting operations to each vertex. In some embodiments, vertex fetcher 3105 and vertex shader 3107 execute vertex-processing instructions by dispatching execution threads to execution units 3152A-3152B via a thread dispatcher 3131.

[0253] In some embodiments, execution units 3152A-3152B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units 3152A-3152B have an attached L1 cache 3151 that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

[0254] In some embodiments, graphics pipeline 3120 includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader 811 configures the tessellation operations. A programmable domain shader 817 provides back-end evaluation of tessellation output. A tessellator 3113 operates at the direction of hull shader 3111 and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline 3120. In some embodiments, if tessellation is not used, tessellation components (e.g., hull shader 3111, tessellator 3113, and domain shader 3117) can be bypassed.

[0255] In some embodiments, complete geometric objects can be processed by a geometry shader 3119 via one or more threads dispatched to execution units 3152A-3152B, or can proceed directly to the clipper 3129. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader 3119 receives input from the vertex shader 3107. In some embodiments, geometry shader 3119 is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

[0256] Before rasterization, a clipper 3129 processes vertex data. The clipper 3129 may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer and depth test component 3173 in the render output pipeline 3170 dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic 3150. In some embodiments, an application can bypass the rasterizer and depth test component 3173 and access un-rasterized vertex data via a stream out unit 3123.

[0257] The graphics processor 3100 has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units 3152A-3152B and associated cache(s) 3151, texture and media sampler 3154, and texture/sampler cache 3158 interconnect via a data port 3156 to perform memory access and communicate with render output pipeline components of the processor. In some embodiments, sampler 3154, caches 3151, 3158 and execution units 3152A-3152B each have separate memory access paths.

[0258] In some embodiments, render output pipeline 3170 contains a rasterizer and depth test component 3173 that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache 3178 and depth cache 3179 are also available in some embodiments. A pixel operations component 3177 performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine 3141, or substituted at display time by the display controller 3143 using overlay display planes. In some embodiments, a shared L3 cache 3175 is available to all graphics components, allowing the sharing of data without the use of main system memory.

[0259] In some embodiments, graphics processor media pipeline 3130 includes a media engine 3137 and a video front-end 3134. In some embodiments, video front-end 3134 receives pipeline commands from the command streamer

3103. In some embodiments, media pipeline 3130 includes a separate command streamer. In some embodiments, video front-end 3134 processes media commands before sending the command to the media engine 3137. In some embodiments, media engine 3137 includes thread spawning functionality to spawn threads for dispatch to thread execution logic 3150 via thread dispatcher 3131.

[0260] In some embodiments, graphics processor 3100 includes a display engine 3140. In some embodiments, display engine 3140 is external to processor 3100 and couples with the graphics processor via the ring interconnect 3102, or some other interconnect bus or fabric. In some embodiments, display engine 3140 includes a 2D engine 3141 and a display controller 3143. In some embodiments, display engine 3140 contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller 3143 couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

[0261] In some embodiments, graphics pipeline 3120 and media pipeline 3130 are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. In some embodiments, support may also be provided for the Direct3D library from the Microsoft Corporation. In some embodiments, a combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming

[0262] **FIG. 32A** is a block diagram illustrating a graphics processor command format 3200 according to some embodiments. **FIG. 32B** is a block diagram illustrating a graphics processor command sequence 3210 according to an embodiment. The solid lined boxes in **FIG. 32A** illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format 3200 of **FIG. 32A** includes data fields to identify a target client 3202 of the command, a command operation code (opcode) 3204, and the relevant data 3206 for the command. A sub-opcode 3205 and a command size 3208 are also included in some commands.

[0263] In some embodiments, client 3202 specifies the client unit of the graphics device that processes the command data. In some embodiments, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In some embodiments, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode 3204 and, if present, sub-opcode 3205 to determine the operation to perform. The client unit performs the command using information in data field 3206. For some commands an explicit command size 3208 is expected to specify the size of the command. In some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

[0264] The flow diagram in **FIG. 32B** shows an exemplary graphics processor command sequence 3210. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

[0265] In some embodiments, the graphics processor command sequence 3210 may begin with a pipeline flush command 3212 to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline 3222 and the media pipeline 3224 do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command 3212 can be used for pipeline synchronization or before placing the graphics processor into a low power state.

[0266] In some embodiments, a pipeline select command 3213 is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some embodiments, a pipeline select command 3213 is

required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command 3212 is required immediately before a pipeline switch via the pipeline select command 3213.

[0267] In some embodiments, a pipeline control command 3214 configures a graphics pipeline for operation and is used to program the 3D pipeline 3222 and the media pipeline 3224. In some embodiments, pipeline control command 3214 configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command 3214 is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

[0268] In some embodiments, commands specific to the return buffer state 3216 are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. In some embodiments, the graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. In some embodiments, the return buffer state 3216 includes selecting the size and number of return buffers to use for a set of pipeline operations.

[0269] The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination 3220, the command sequence is tailored to the 3D pipeline 3222 beginning with the 3D pipeline state 3230 or the media pipeline 3224 beginning at the media pipeline state 3240.

[0270] The commands to configure the 3D pipeline state 3230 include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. In some embodiments, 3D pipeline state 3230 commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

[0271] In some embodiments, 3D primitive 3232 command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive 3232 command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive 3232 command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive 3232 command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline 3222 dispatches shader execution threads to graphics processor execution units.

[0272] In some embodiments, 3D pipeline 3222 is triggered via an execute 3234 command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment, command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

[0273] In some embodiments, the graphics processor command sequence 3210 follows the media pipeline 3224 path when performing media operations. In general, the specific use and manner of programming for the media pipeline 3224 depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

[0274] In some embodiments, media pipeline 3224 is configured in a similar manner as the 3D pipeline 3222. A set of commands to configure the media pipeline state 3240 are dispatched or placed into a command queue before the media object commands 3242. In some embodiments, media pipeline state commands 3240 include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands 3240 also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

[0275] In some embodiments, media object commands 3242 supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command 3242. Once the pipeline state is configured and media object commands 3242 are queued, the media pipeline 3224 is triggered via an execute command 3244 or an equivalent execute event (e.g., register write). Output from media pipeline 3224 may then be post processed by operations provided by the 3D pipeline 3222 or the media pipeline 3224. In some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

Graphics Software Architecture

[0276] **FIG. 33** illustrates exemplary graphics software architecture for a data processing system 3300 according to some embodiments. In some embodiments, software architecture includes a 3D graphics application 3310, an operating system 3320, and at least one processor 3330. In some embodiments, processor 3330 includes a graphics processor 3332 and one or more general-purpose processor core(s) 3334. The graphics application 3310 and operating system 3320 each execute in the system memory 3350 of the data processing system.

[0277] In some embodiments, 3D graphics application 3310 contains one or more shader programs including shader instructions 3312. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions 3314 in a machine language suitable for execution by the general-purpose processor core 3334. The application also includes graphics objects 3316 defined by vertex data.

[0278] In some embodiments, operating system 3320 is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system 3320 can support a graphics API 3322 such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system 3320 uses a front-end shader compiler 3324 to compile any shader instructions 3312 in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application 3310. In some embodiments, the shader instructions 3312 are provided in an intermediate form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

[0279] In some embodiments, user mode graphics driver 3326 contains a back-end shader compiler 3327 to convert the shader instructions 3312 into a hardware specific representation. When the OpenGL API is in use, shader instructions 3312 in the GLSL high-level language are passed to a user mode graphics driver 3326 for compilation. In some embodiments, user mode graphics driver 3326 uses operating system kernel mode functions 3328 to communicate with a kernel mode graphics driver 3329. In some embodiments, kernel mode graphics driver 3329 communicates with graphics processor 3332 to dispatch commands and instructions.

IP Core Implementations

[0280] One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

[0281] **FIG. 34** is a block diagram illustrating an IP core development system 3400 that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system 3400 may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility 3430 can generate a software simulation 3410 of an IP core design in a high level programming language (e.g., C/C++). The software simulation 3410 can be used to design, test, and verify the behavior of the IP core using a simulation model 3412. The simulation model 3412 may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design 3415 can then be created or synthesized from the simulation model 3412. The RTL design 3415 is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design 3415, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

[0282] The RTL design 3415 or equivalent may be further synthesized by the design facility into a hardware model 3’0, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility 3465 using non-volatile memory 3440 (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection 3450 or wireless connection 3460. The fabrication facility 3465 may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

Exemplary System on a Chip Integrated Circuit

[0283] **FIGs. 35-37** illustrated exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores, according to various embodiments described herein. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

[0284] **FIG. 35** is a block diagram illustrating an exemplary system on a chip integrated circuit 3500 that may be fabricated using one or more IP cores, according to an embodiment. Exemplary integrated circuit 3500 includes one or more application processor(s) 3505 (e.g., CPUs), at least one graphics processor 3510, and may additionally include an image processor 3515 and/or a video processor 3520, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit 3500 includes peripheral or bus logic including a USB controller 3525, UART controller 3530, an SPI/SDIO controller 3535, and an I^2S/I^2C controller 3540. Additionally, the integrated circuit can include a display device 3545 coupled to one or more of a high-definition multimedia interface (HDMI) controller 3550 and a mobile industry processor interface (MIPI) display interface 3555. Storage may be provided by a flash memory subsystem 3560 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller 3565 for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine 3570.

[0285] **FIG. 36** is a block diagram illustrating an exemplary graphics processor 3610 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 3610 can be a variant of the graphics processor 3610 of **FIG. 36.** Graphics processor 3610 includes a vertex processor 3605 and one or more fragment processor(s) 3615A-3615N (e.g., 3615A, 3615B, 3615C, 3615D, through 3615N-1, and 3615N). Graphics processor 3610 can execute different shader programs via separate logic, such that the vertex processor 3605 is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) 3615A-3615N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor 3605 performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) 3615A-3615N use the primitive and vertex data generated by the vertex processor 3605 to produce a framebuffer that is displayed on a display device. In one embodiment, the fragment processor(s) 3615A-3615N are optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

[0286] Graphics processor 3610 additionally includes one or more memory management units (MMUs) 3620A-3620B, cache(s) 3625A-3625B, and circuit interconnect(s) 3630A-3630B. The one or more MMU(s) 3620A-3620B provide for virtual to physical address mapping for integrated circuit 3610, including for the vertex processor 3605 and/or fragment processor(s) 3615A-3615N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) 3625A-3625B. In one embodiment the one or more MMU(s) 3625A-3625B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) 3605, image processor 3615, and/or video processor 3620 of **FIG. 36,** such that each processor 3605-3620 can participate in a shared or unified virtual memory system. The one or more circuit interconnect(s) 3630A-3630B enable graphics processor 3610 to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments.

[0287] **FIG. 37** is a block diagram illustrating an additional exemplary graphics processor 3710 of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor 3710 can be a variant of the graphics processor 3510 of **FIG. 35.** Graphics processor 3710 includes the one or more MMU(s) 3520A-3520B, caches 3525A-3525B, and circuit interconnects 3530A-3530B of the integrated circuit 3500 of **FIG. 35.**

[0288] Graphics processor 3710 includes one or more shader core(s) 3715A-3715N (e.g., 3715A, 3715B, 3715C, 3715D, 3715E, 3715F, through 3715N-1, and 3715N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor 3710 includes an inter-core task manager 3705, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores 3715A-3715N and a tiling unit 3718 to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches.

[0289] Embodiments described herein provide a logic unit that includes a merged integer/floating-point datapath for both multiply-add (e.g., a*b+c) and multiply-accumulate (e.g., c=c+a*b) operations. In one embodiment an addend for the add operation is based on an accumulation of previous operations. In one embodiment the integer datapath of the logic unit is merged into a floating-point datapath that has addend alignment operation in parallel with a multiply operation. In one embodiment the integer datapath is merged into a floating-point datapath that has addend alignment operation

after the multiply operation. The multiply-add and multiply-accumulate datapaths described herein may be single-cycle or multi-cycle.

[0290] In one embodiment, during a two-cycle floating-point multiply-accumulate, the logic unit does not compare the mantissas at the beginning of second stage (e.g., adder stage). Instead, the logic unit precomputes a larger (or smaller) mantissa based on an accumulator exponent from the second stage and multiplier output computed during the first stage.

[0291] In one embodiment the accumulator or addend mantissa bit-width is larger than the mantissa bit-widths of the multiplier inputs. In one embodiment the integer operations are mapped on to a floating-point unit. Some of the integer operations are also mapped onto existing exponent circuits in addition to the mantissa circuits of a floating-point unit. In one embodiment the logic units described herein include a multiplier unit and an adder unit that are shared between floating-point and integer operations and are used to perform both floating-point and integer operations.

[0292] The following clauses and/or examples pertain to specific embodiments or examples thereof. Specifics in the examples may be used anywhere in one or more embodiments. The various features of the different embodiments or examples may be variously combined with some features included and others excluded to suit a variety of different applications. Examples may include subject matter such as a method, means for performing acts of the method, at least one machine-readable medium including instructions that, when performed by a machine cause the machine to perform acts of the method, or of an apparatus or system according to embodiments and examples described herein. Various components can be a means for performing the operations or functions described.

[0293] One embodiment provides for a machine-learning hardware accelerator comprising a compute unit having an adder and a multiplier that are shared between integer data path and a floating-point datapath, the upper bits of input operands to the multiplier to be gated during floating-point operation. In one embodiment the adder and the multiplier are configurable to perform floating-point operation and integer operation. In one embodiment the compute unit is to perform a multiply-add operation via the multiplier and the adder. In one embodiment the compute unit accepts at least two input operands. One embodiment provides for a compute unit to perform a multiply-accumulate operation using two-input operands and an accumulated value. One embodiment provides for a compute unit to perform a multiply-add operation using three input operands. In one embodiment the compute unit is to perform a multiply-accumulate operation or a multiply-add operation within a single cycle. In one embodiment the compute unit is to perform a two-cycle multiply-add operation or a two-cycle multiply-accumulate operation. In one embodiment the multiplier within the compute unit is to produce an output during a first cycle and the adder is to produce an output during a second cycle. In one embodiment the compute unit is to perform a two-cycle multiply-accumulate operation in which the first cycle is associated with a first logic stage, the second cycle is associated with a second logic stage, and the compute unit includes an exponent unit to precompute a larger mantissa and alignment shift for the second stage via an accumulated output of a previous cycle of the second stage and multiplier output from the first stage.

[0294] In one embodiment the integer datapath is merged into the floating-point datapath having an addend alignment operation in parallel with the multiply operation. In one embodiment the integer datapath is merged into the floating-point datapath having an addend alignment operation after the multiply operation. The compute unit can have a mode input to switch the compute unit between integer operation and floating-point operation. In one embodiment the compute unit is configurable for an 8.8 fixed-point input and a 16.0 fixed-point output.

[0295] One embodiment provides for a data processing system comprising a non-transitory machine-readable medium to store instructions for execution by one or more processors of the data processing system; and a general-purpose graphics processing unit comprising a machine-learning hardware accelerator and a dynamic precision compute unit, the machine-learning hardware accelerator including hardware logic to perform multiple machine-learning compute operations in response to a single instruction. In one embodiment the dynamic precision compute unit is switchable between integer operation and floating-point operation. In one embodiment the dynamic precision compute unit includes an integer datapath and floating-point datapath that share a multiplier and an adder, where the multiplier is to perform a multiply operation for the integer datapath and the floating-point datapath. In one embodiment the floating-point datapath includes an addend alignment operation performed in parallel with the multiply operation. In one embodiment the floating-point datapath includes an addend alignment operation performed after the multiply operation. In one embodiment the dynamic precision compute unit is configured for a single-cycle fused multiply-add operation or a two-cycle fused multiply-accumulate operation.

[0296] One embodiment provides for a method of accelerating machine-learning operations, the method comprising fetching and decoding a single instruction to perform a combined multiply and add operation on a set of operands; issuing the single instruction for execution by a dynamically configurable compute unit; configuring one of more logic units of the compute unit to perform operations at the precision and data-type of the set of operands; and executing at least a portion of the single instruction at the dynamically configurable compute unit to generate and output based on the multiply and add operation.

[0297] The embodiments described herein refer to specific configurations of hardware, such as application specific integrated circuits (ASICs), configured to perform certain operations or having a predetermined functionality. Such electronic devices typically include a set of one or more processors coupled to one or more other components, such as

one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage devices of a given electronic device typically store code and/or data for execution on the set of one or more processors of that electronic device.

[0298] Of course, one or more parts of an embodiment may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the embodiments may be practiced without some of these specific details. In certain instances, well-known structures and functions were not described in elaborate detail to avoid obscuring the inventive subject matter of the embodiments. Accordingly, the scope and spirit of the invention should be judged in terms of the claims that follow.

Claims

1. An apparatus comprising:

a memory interface;
an array of processing clusters each including a multiprocessor unit coupled to the memory interface,
at least one multiprocessor unit to execute a fused multiply-add instruction in parallel across multiple threads,
the at least one multiprocessor unit comprising:

a register file to store data; and
a compute unit coupled to the register file,
the compute unit to execute a fused multiply-add instruction on matrix data,
the compute unit comprising:

hardware logic to quantize the data from a higher precision, including a 32-bit floating point format to a lower precision floating point format, including a 16-bit floating point format, having a 1-bit sign, an 8-bit exponent, and a mantissa,
wherein fewer bits are used for the mantissa of the lower precision floating point format; and
one or more logic units to perform the fused multiply-add operation on the data in the lower precision floating point format.

2. Apparatus according to claim 1, wherein the matrix data comprises a first plurality of data elements of a first multi-dimensional source matrix and a second plurality of data elements of a second multidimensional source matrix.

3. Apparatus according to claim 2, wherein the one or more logic units perform parallel fused multiply add operations to multiply the first plurality of data elements in the lower precision floating point format by corresponding data elements of the second plurality of data elements in the lower precision floating point format
to generate a plurality of products and
to add the products to corresponding 32-bit floating point values
to generate corresponding 32-bit floating point data elements of a multidimensional result matrix.

4. Apparatus according to one or more of claims 1-3, wherein the mantissa of the lower precision floating point format comprises a 7-bit mantissa.

5. Apparatus according to one or more of claims 1-4, wherein the fused multiply-add instruction comprises a first operand to identify the first plurality of data elements and a second operand to identify the second plurality of data elements.

6. Apparatus according to claim 5, wherein
the first operand identifies the first plurality of data elements in a first one or more registers of the register file and
the second operand identifies the second plurality of data elements in a second one or more registers of the register file.

7. Apparatus according to one or more of claims 1-3, further comprising:

an instruction fetch unit to fetch the fused multiply-add instruction;
a decoder to decode the fused multiply-add instruction to generate parallel multiply-add operations to be performed across multiple threads; and
a scheduler to schedule the parallel multiply-add operations for execution by the compute unit.

8. Apparatus according to claim 7, wherein the compute unit comprises a plurality of arithmetic logic units, ALUs.

9. Apparatus according to one or more of claims 1-3, further comprising:
an interconnect fabric to couple the array of processing clusters to the memory interface.

10. Apparatus according to claim 9, further comprising:
an input/output, I/O, interface coupled to the interconnect fabric.

11. Apparatus according to claim 10, further comprising:
a shared cache or local memory shared by the one or more logic units.

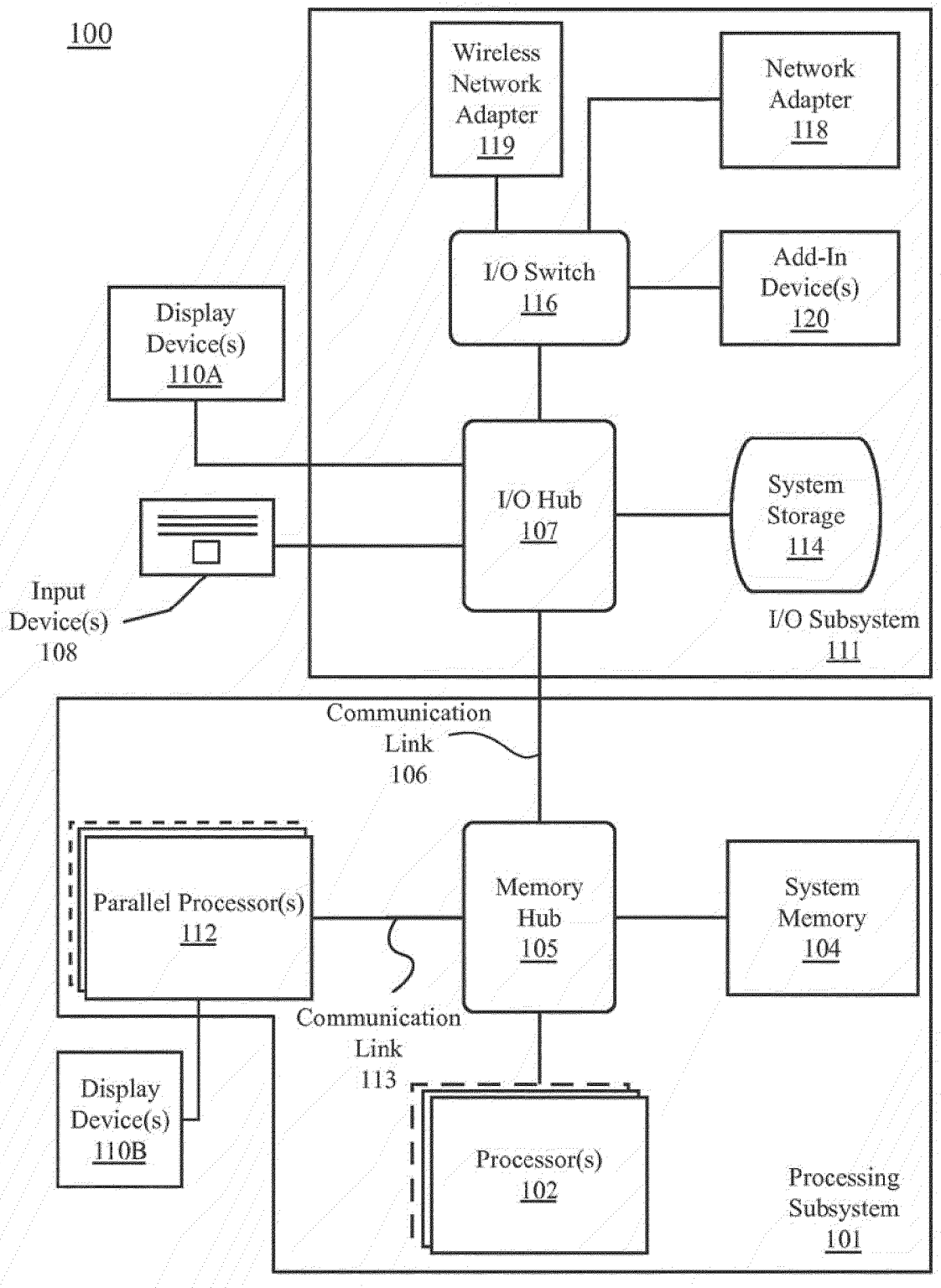

100
Wireless Network Adapter 119
Network Adapter 118
I/O Switch 116
Add-In Device(s) 120
Display Device(s) 110A
I/O Hub 107
System Storage 114
Input Device(s) 108
I/O Subsystem 111
Communication Link 106
Parallel Processor(s) 112
Memory Hub 105
System Memory 104
Communication Link 113
Display Device(s) 110B
Processor(s) 102
Processing Subsystem 101

FIG. 1

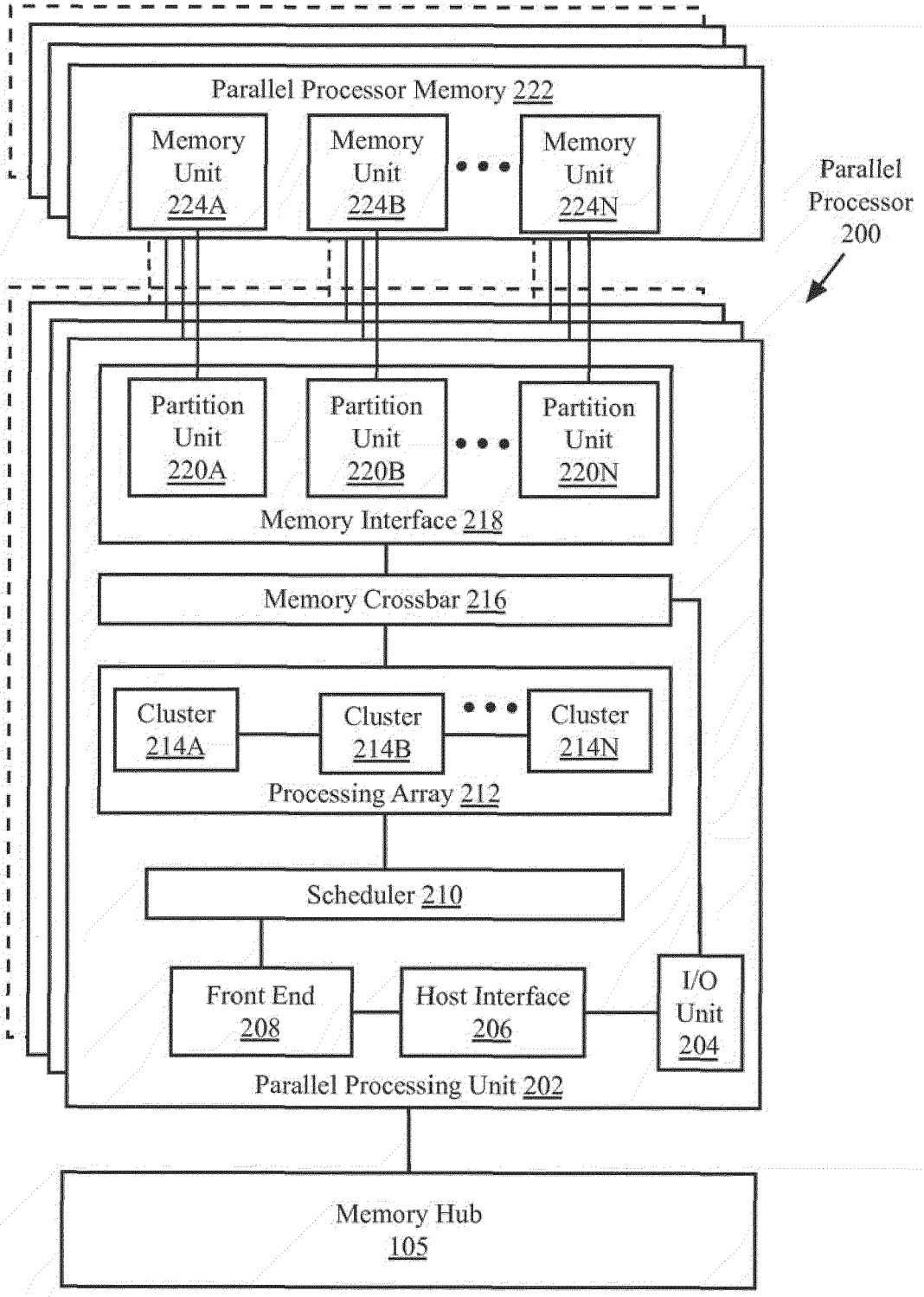

Parallel Processor Memory 222
Memory Unit 224A
Memory Unit 224B
Memory Unit 224N
Parallel Processor 200
Partition Unit 220A
Partition Unit 220B
Partition Unit 220N
Memory Interface 218
Memory Crossbar 216
Cluster 214A
Cluster 214B
Cluster 214N
Processing Array 212
Scheduler 210
Front End 208
Host Interface 206
I/O Unit 204
Parallel Processing Unit 202
Memory Hub 105

FIG. 2A

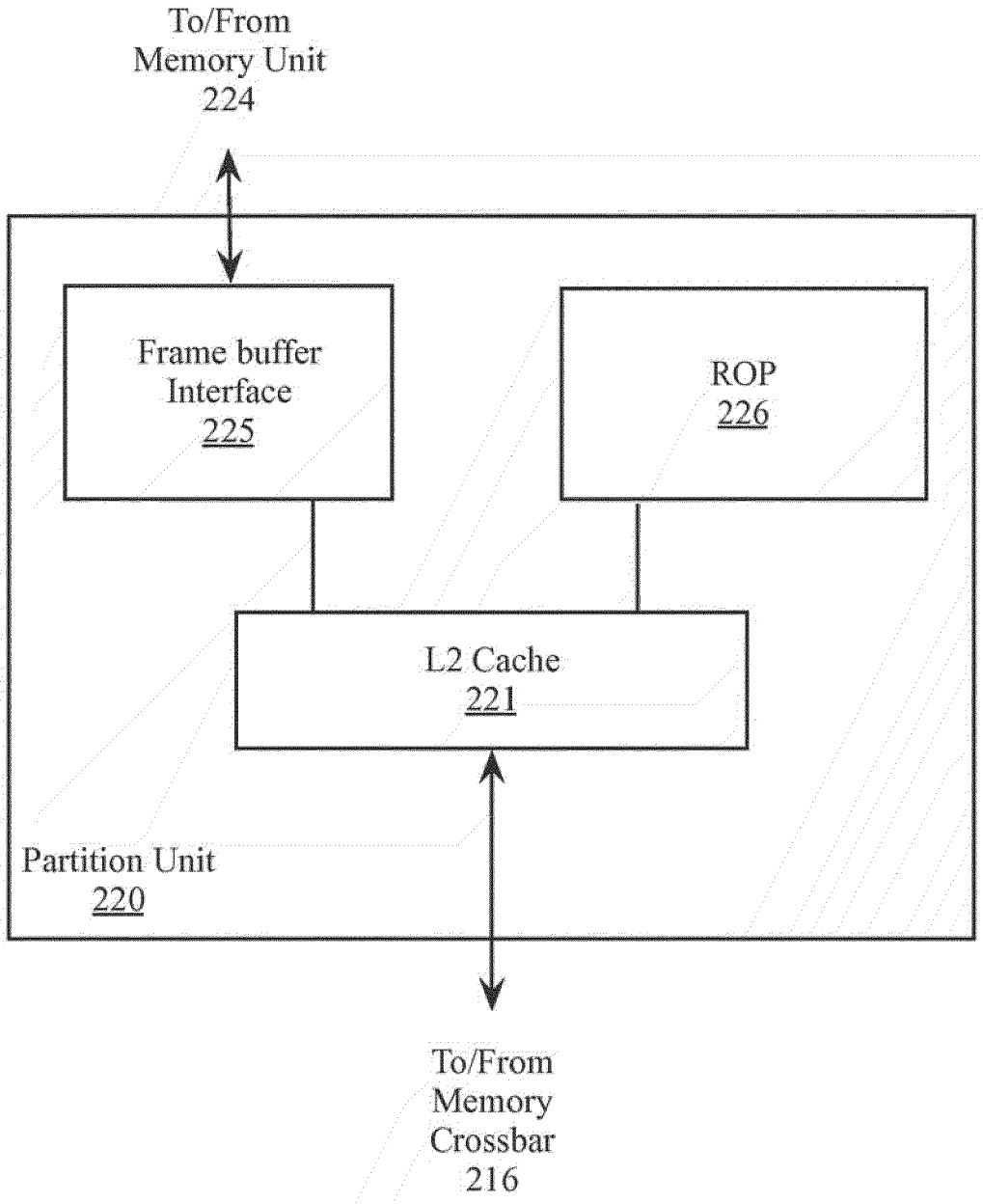

To/From
Memory Unit
224
Frame buffer
Interface
225
ROP
226
L2 Cache
221
Partition Unit
220
To/From
Memory
Crossbar
216

FIG. 2B

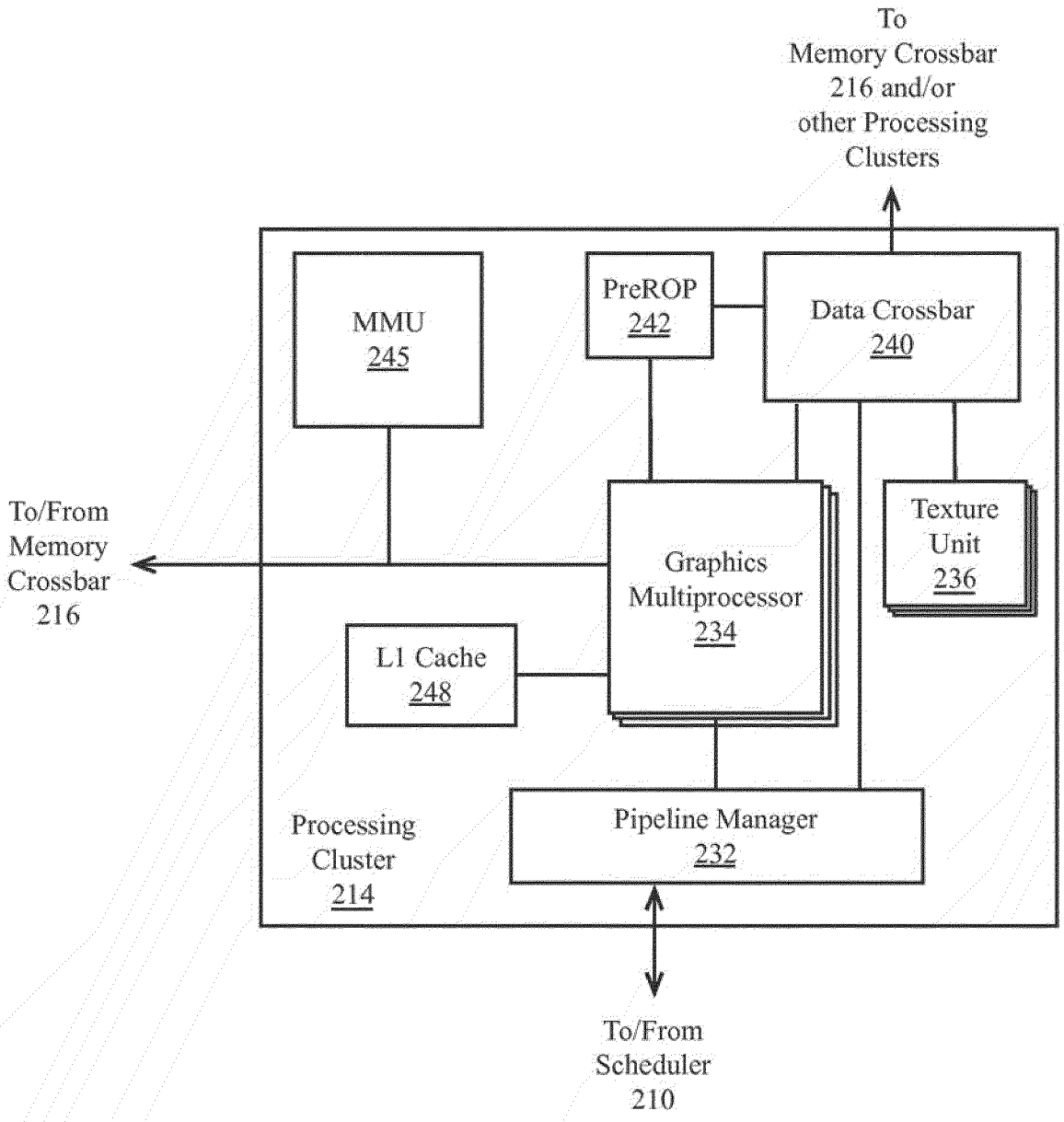

To
Memory Crossbar
216 and/or
other Processing
Clusters
MMU
245
PreROP
242
Data Crossbar
240
To/From
Memory
Crossbar
216
Graphics
Multiprocessor
234
Texture
Unit
236
L1 Cache
248
Processing
Cluster
214
Pipeline Manager
232
To/From
Scheduler
210

FIG. 2C

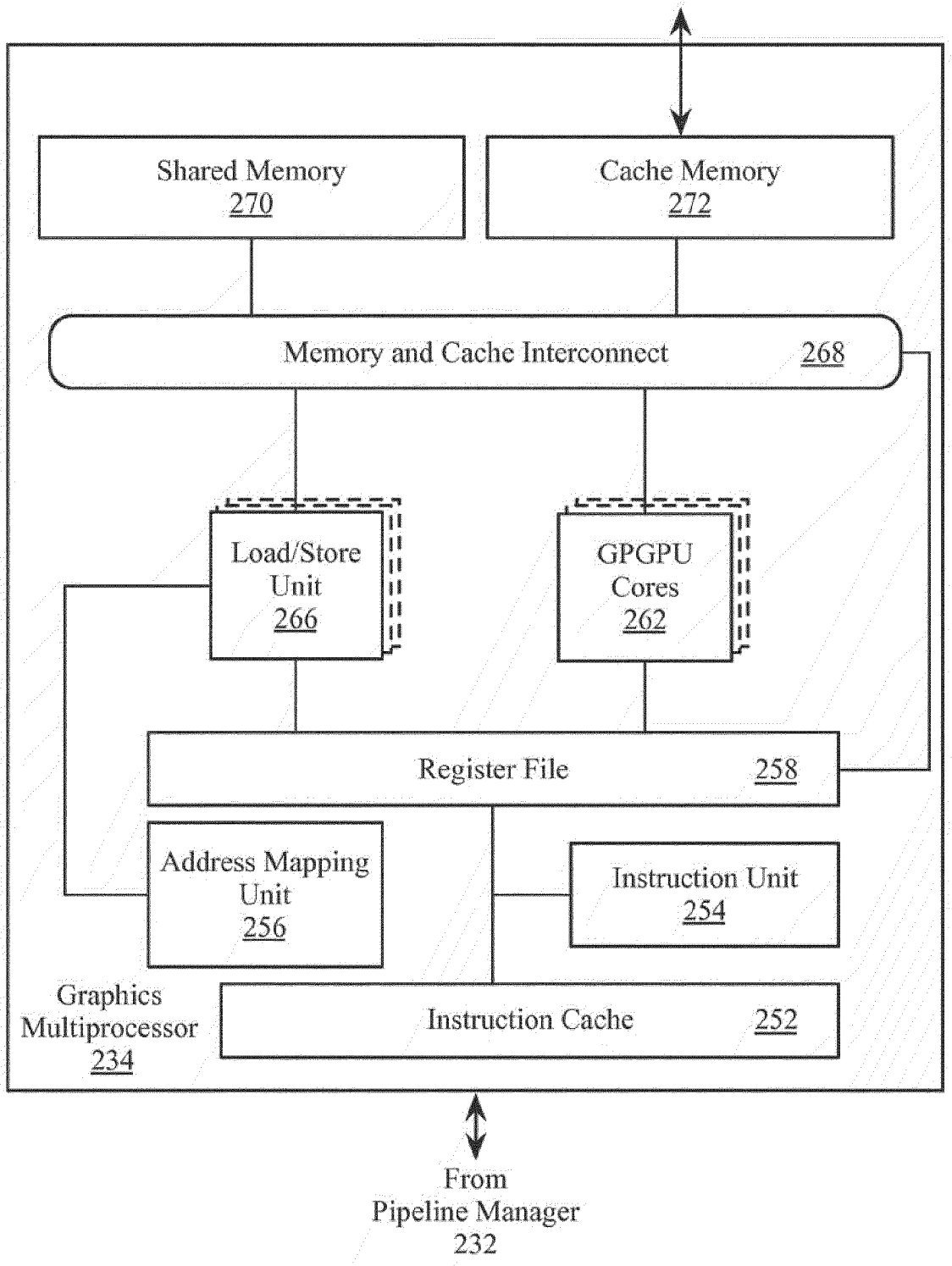

Shared Memory
270
Cache Memory
272
Memory and Cache Interconnect
268
Load/Store
Unit
266
GPGPU
Cores
262
Register File
258
Address Mapping
Unit
256
Instruction Unit
254
Graphics
Multiprocessor
234
Instruction Cache
252
From
Pipeline Manager
232

FIG. 2D

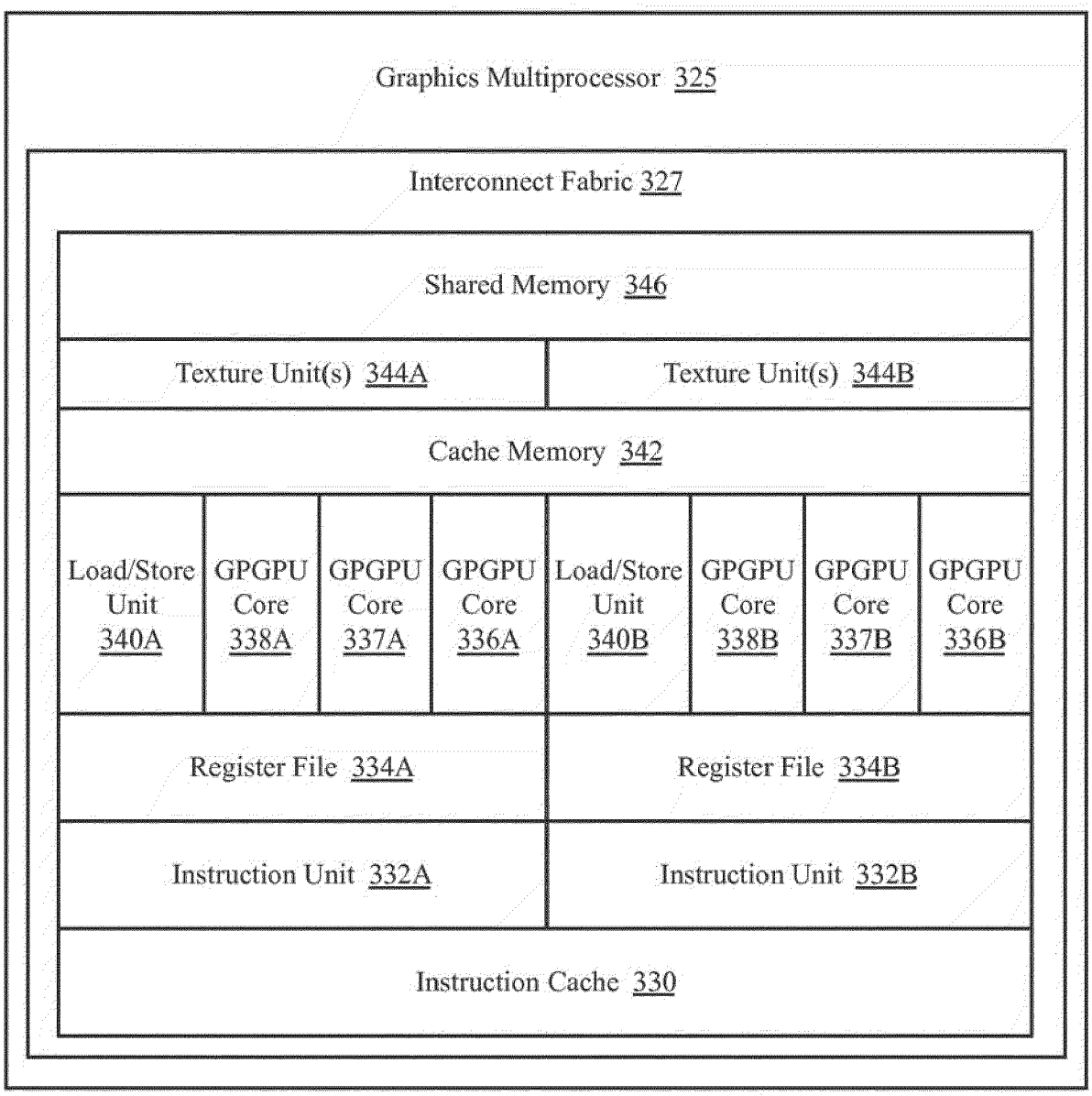

Graphics Multiprocessor 325
Interconnect Fabric 327
Shared Memory 346
Texture Unit(s) 344A
Texture Unit(s) 344B
Cache Memory 342
Load/Store Unit 340A
GPGPU Core 338A
GPGPU Core 337A
GPGPU Core 336A
Load/Store Unit 340B
GPGPU Core 338B
GPGPU Core 337B
GPGPU Core 336B
Register File 334A
Register File 334B
Instruction Unit 332A
Instruction Unit 332B
Instruction Cache 330

FIG. 3A

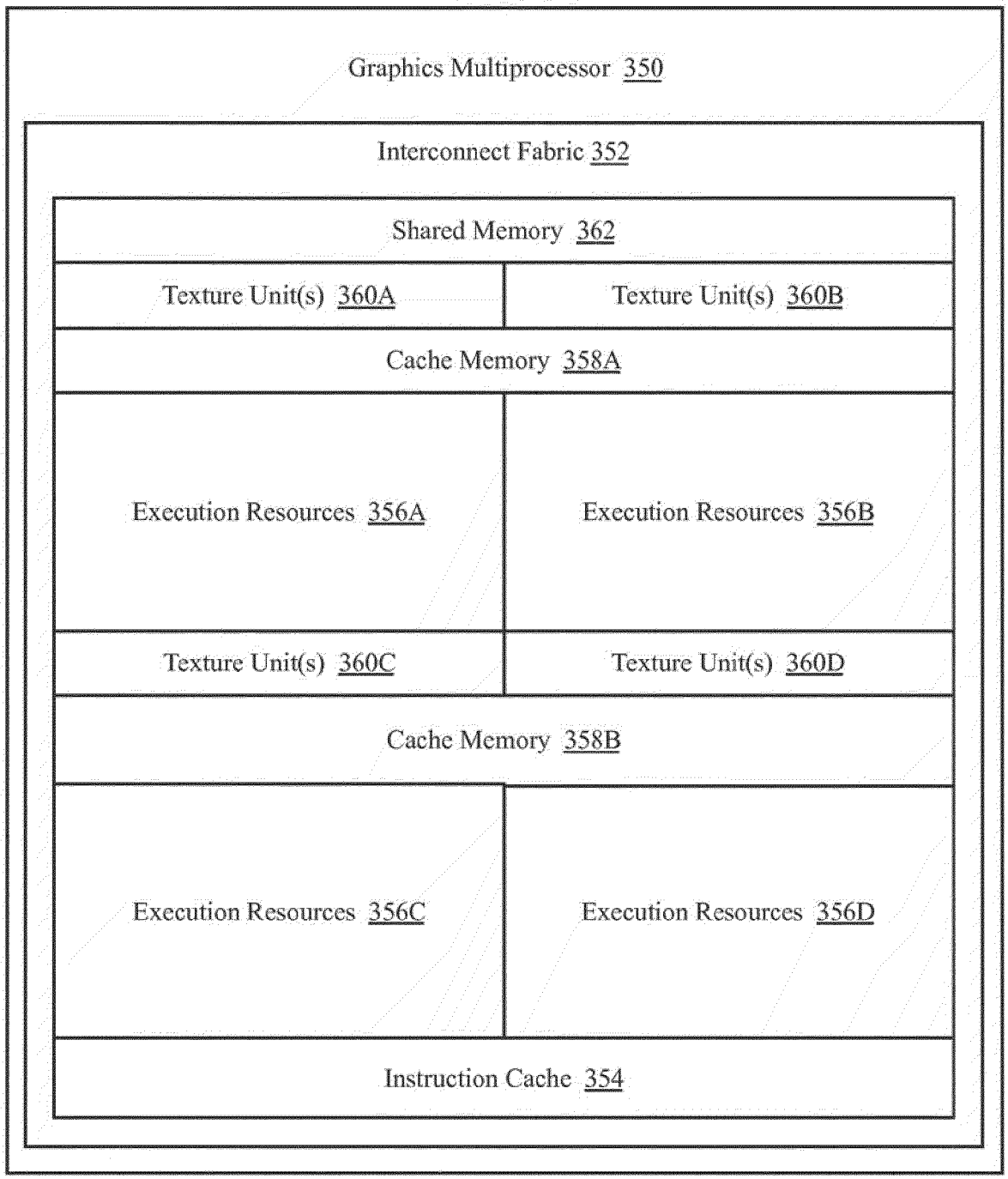

Graphics Multiprocessor 350
Interconnect Fabric 352
Shared Memory 362
Texture Unit(s) 360A
Texture Unit(s) 360B
Cache Memory 358A
Execution Resources 356A
Execution Resources 356B
Texture Unit(s) 360C
Texture Unit(s) 360D
Cache Memory 358B
Execution Resources 356C
Execution Resources 356D
Instruction Cache 354

FIG. 3B

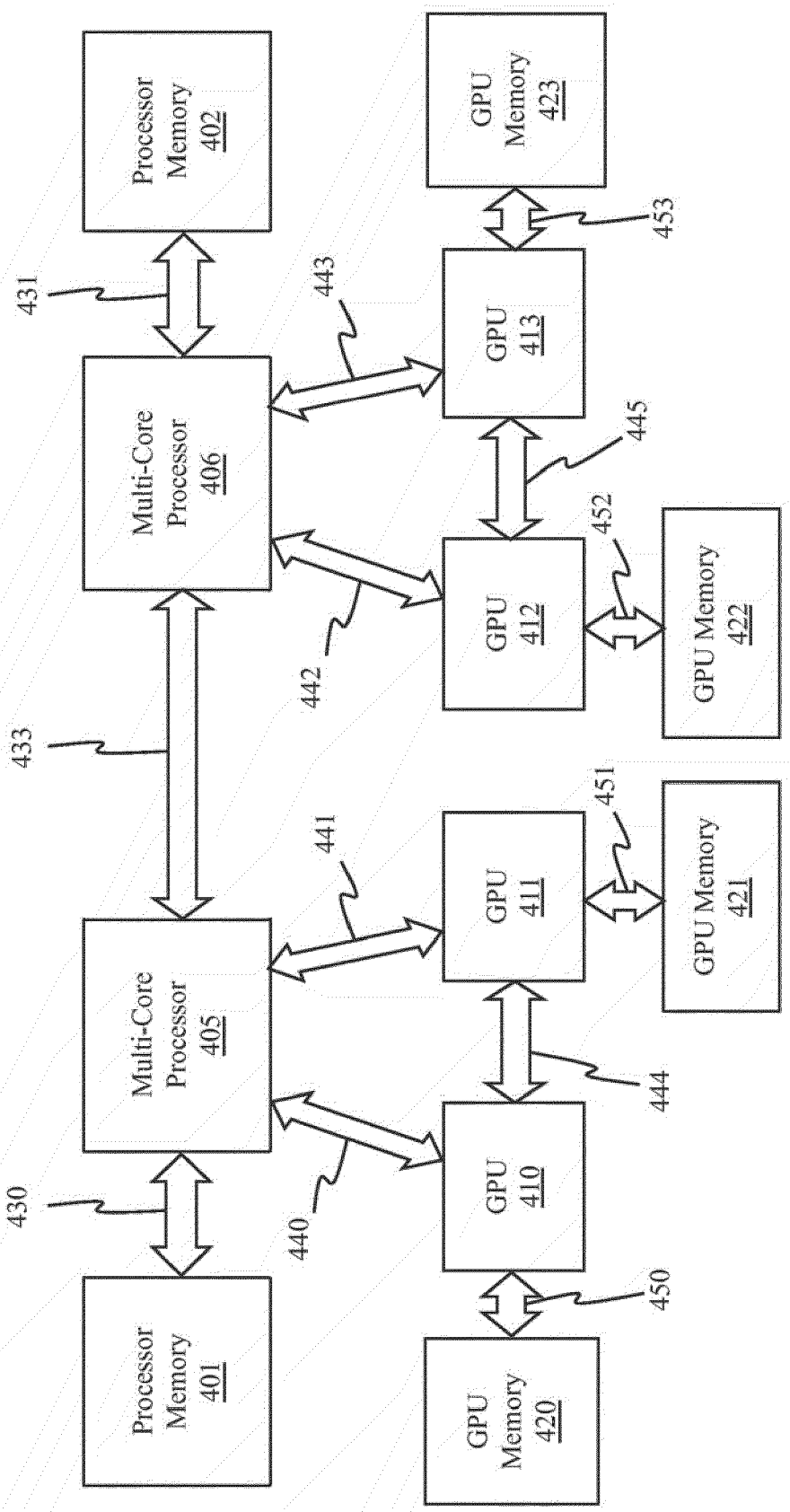

Processor Memory 402
GPU Memory 423
431
443
453
Multi-Core Processor 406
GPU 413
445
452
GPU 412
GPU Memory 422
442
433
441
451
Multi-Core Processor 405
GPU 411
GPU Memory 421
444
430
440
GPU 410
450
Processor Memory 401
GPU Memory 420

FIG. 4A

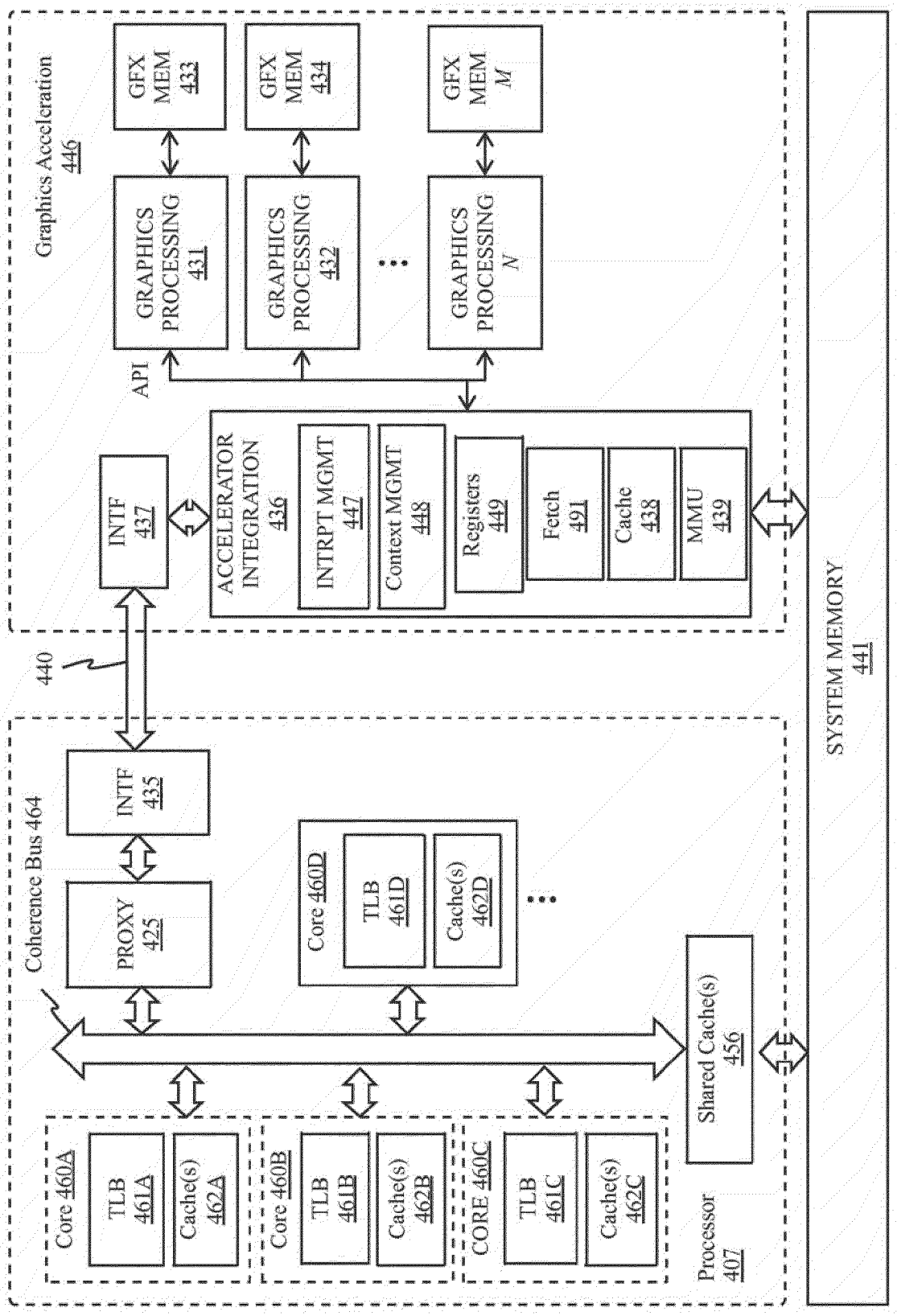

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING 431
GRAPHICS PROCESSING 432
GRAPHICS PROCESSING N
API
INTF 437
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Registers 449
Fetch 491
Cache 438
MMU 439
440
Coherence Bus 464
INTF 435
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Shared Cache(s) 456
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 460C
TLB 461C
Cache(s) 462C
Processor 407
SYSTEM MEMORY 441

FIG. 4B

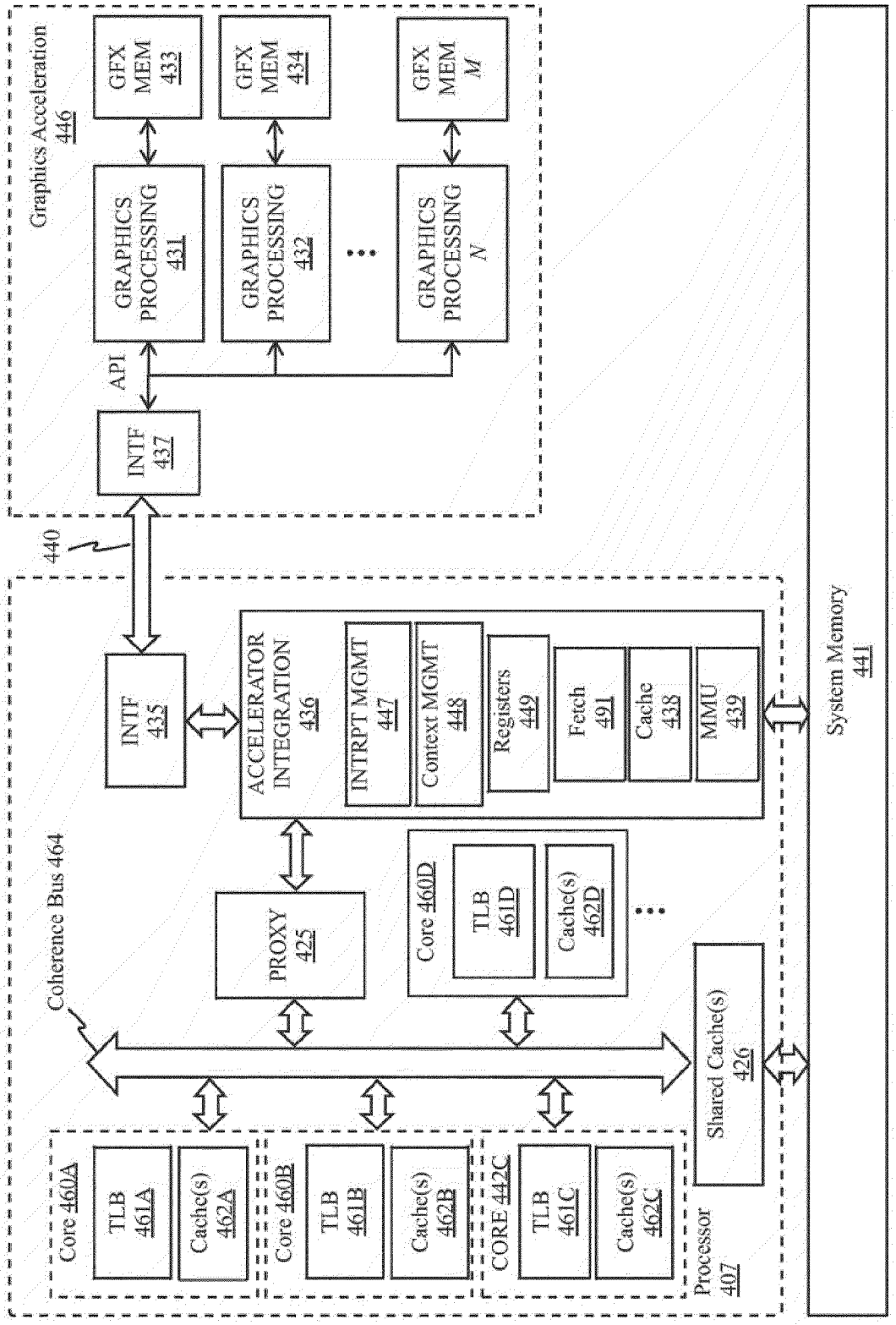

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING 431
GRAPHICS PROCESSING 432
GRAPHICS PROCESSING N
API
INTF 437
440
INTF 435
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Registers 449
Fetch 491
Cache 438
MMU 439
System Memory 441
Coherence Bus 464
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Shared Cache(s) 426
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 442C
TLB 461C
Cache(s) 462C
Processor 407

FIG. 4C

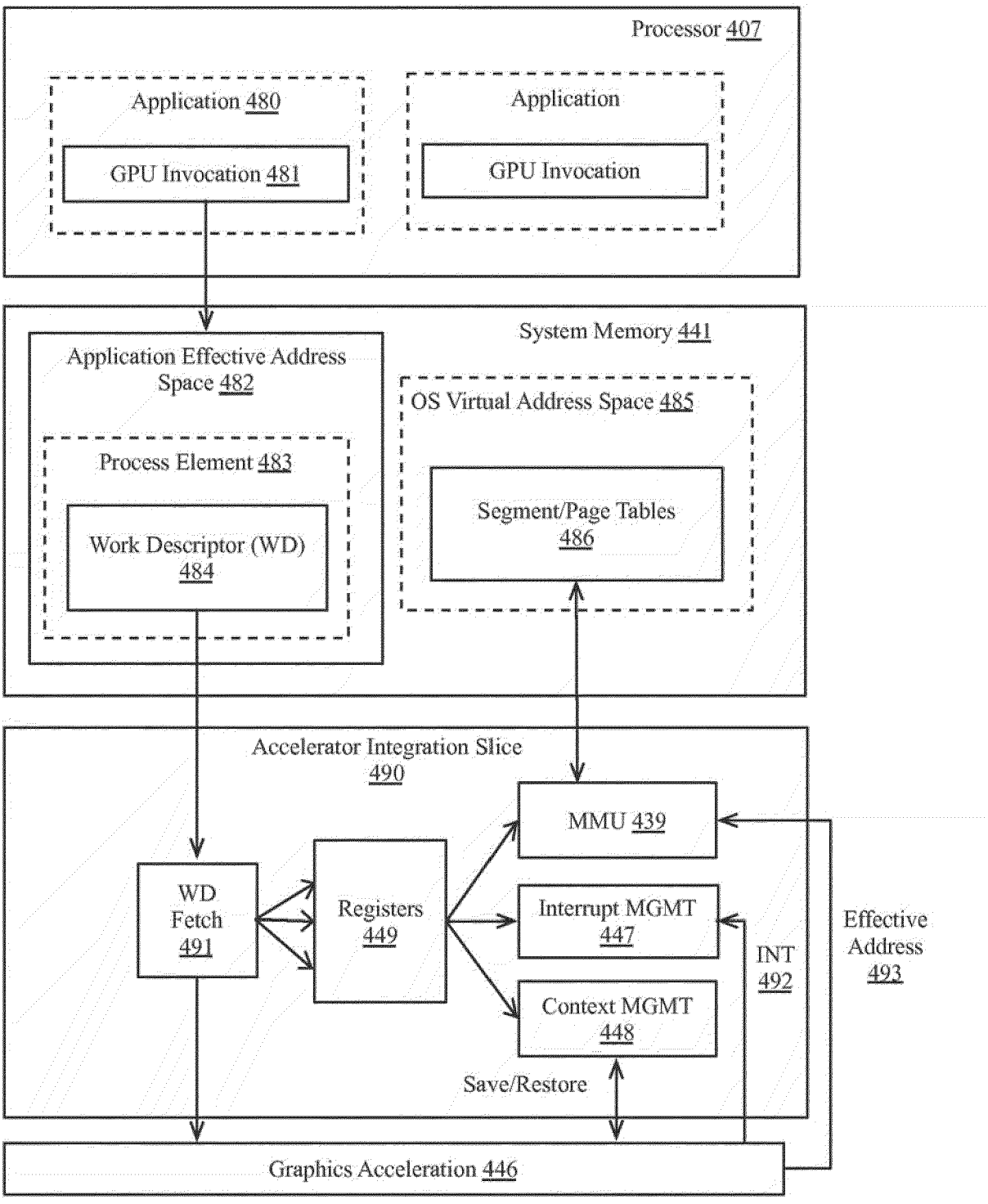

Processor 407
Application 480
GPU Invocation 481
Application
GPU Invocation
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Accelerator Integration Slice 490
MMU 439
WD Fetch 491
Registers 449
Interrupt MGMT 447
Context MGMT 448
Save/Restore
INT 492
Effective Address 493
Graphics Acceleration 446

FIG. 4D

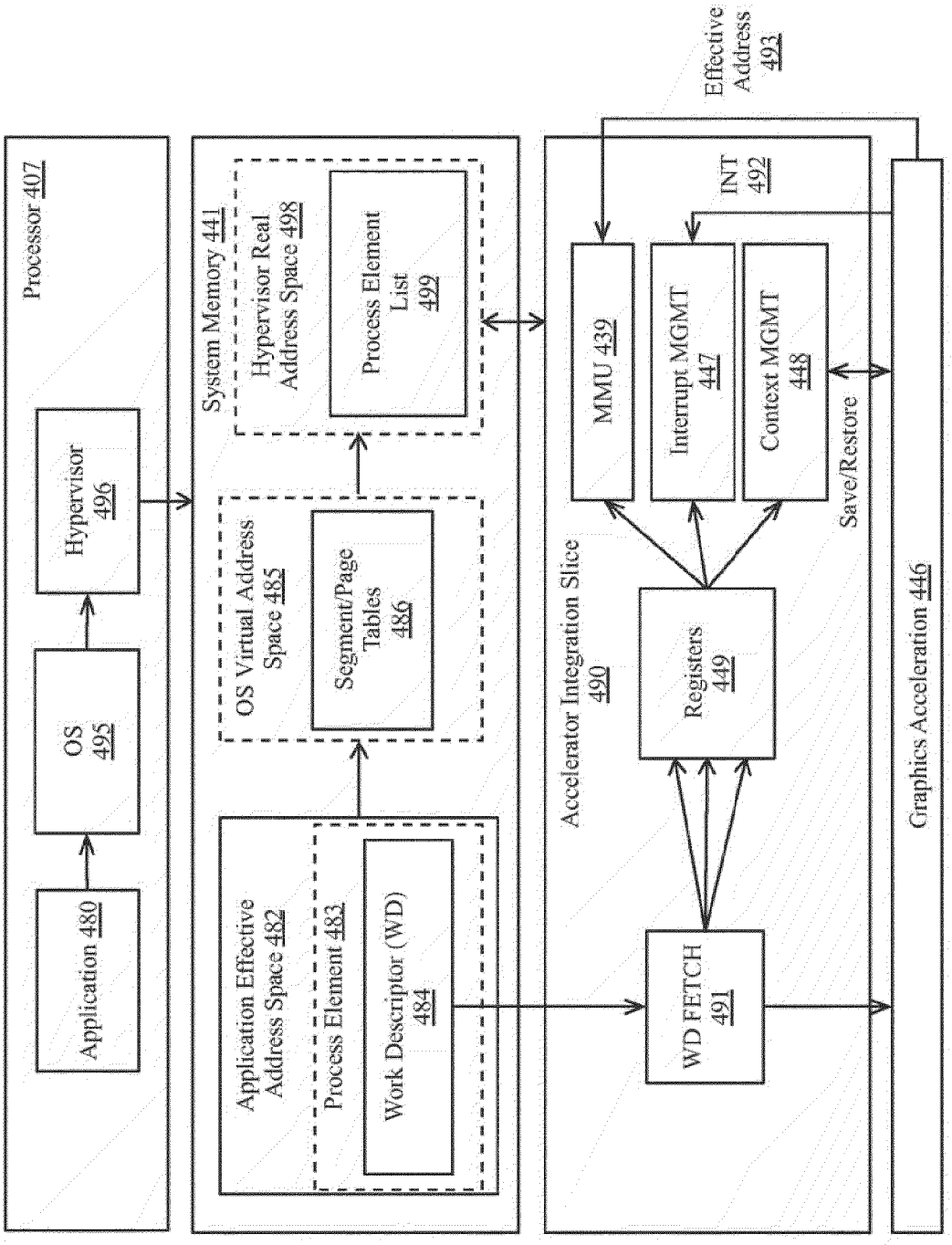

Processor 407
Application 480
OS 495
Hypervisor 496
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Hypervisor Real Address Space 498
Process Element List 499
Accelerator Integration Slice 490
WD FETCH 491
Registers 449
MMU 439
Interrupt MGMT 447
Context MGMT 448
Save/Restore
INT 492
Effective Address 493
Graphics Acceleration 446

FIG. 4E

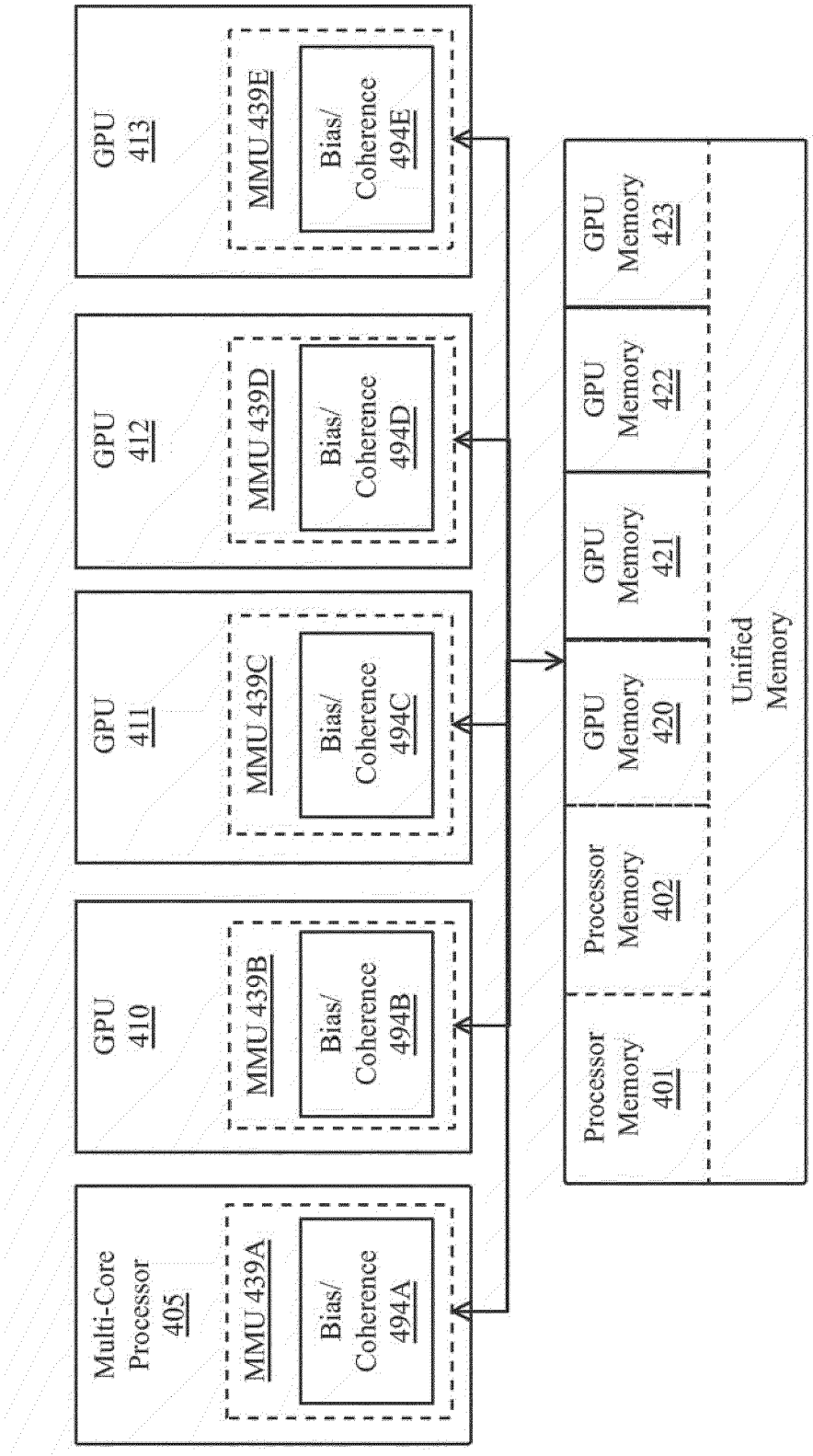

Multi-Core Processor 405
GPU 410
GPU 411
GPU 412
GPU 413
MMU 439A
MMU 439B
MMU 439C
MMU 439D
MMU 439E
Bias/ Coherence 494A
Bias/ Coherence 494B
Bias/ Coherence 494C
Bias/ Coherence 494D
Bias/ Coherence 494E
Processor Memory 401
Processor Memory 402
GPU Memory 420
GPU Memory 421
GPU Memory 422
GPU Memory 423
Unified Memory

FIG. 4F

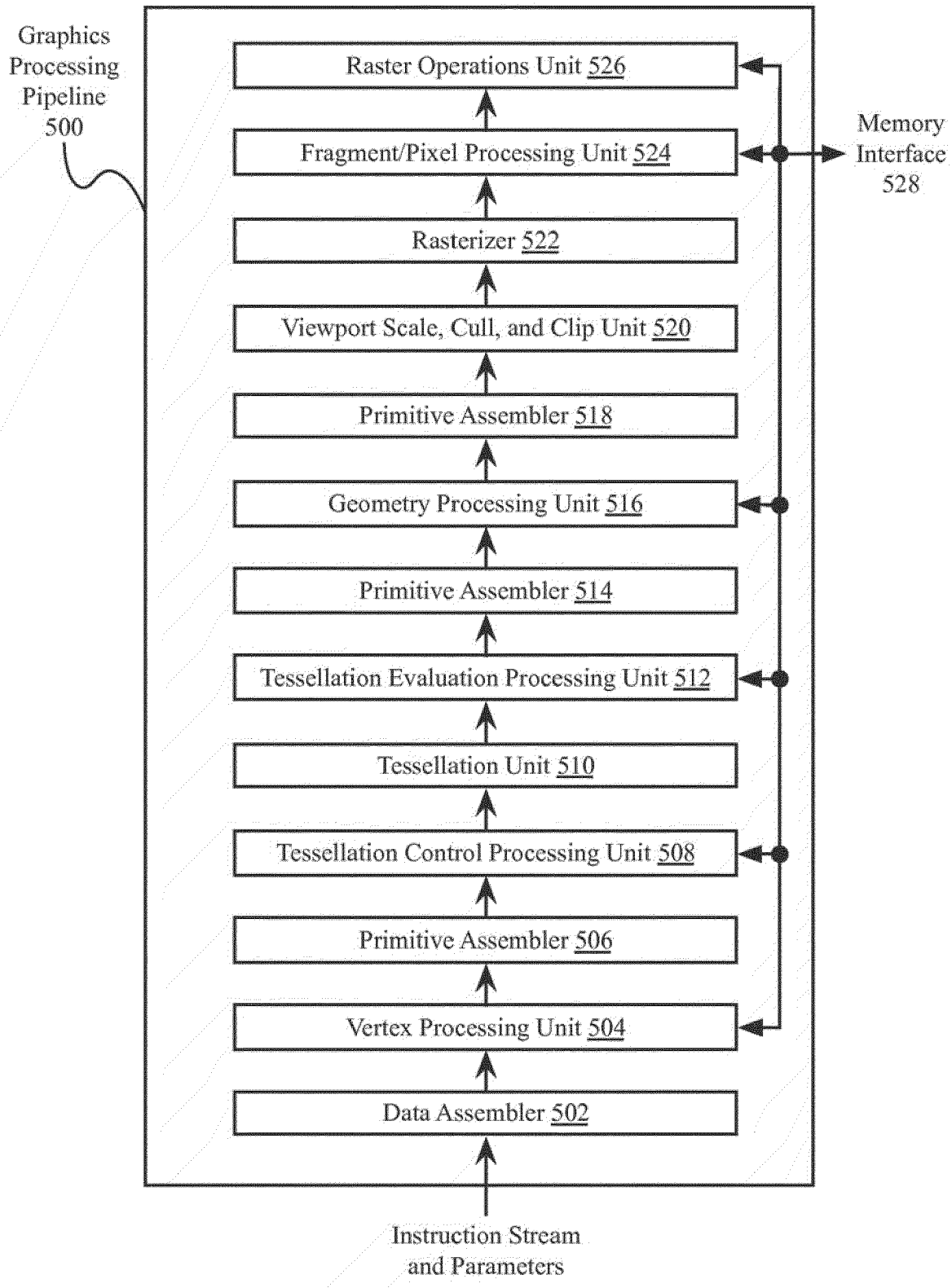

Graphics Processing Pipeline 500
Raster Operations Unit 526
Fragment/Pixel Processing Unit 524
Rasterizer 522
Viewport Scale, Cull, and Clip Unit 520
Primitive Assembler 518
Geometry Processing Unit 516
Primitive Assembler 514
Tessellation Evaluation Processing Unit 512
Tessellation Unit 510
Tessellation Control Processing Unit 508
Primitive Assembler 506
Vertex Processing Unit 504
Data Assembler 502
Memory Interface 528
Instruction Stream and Parameters

FIG. 5

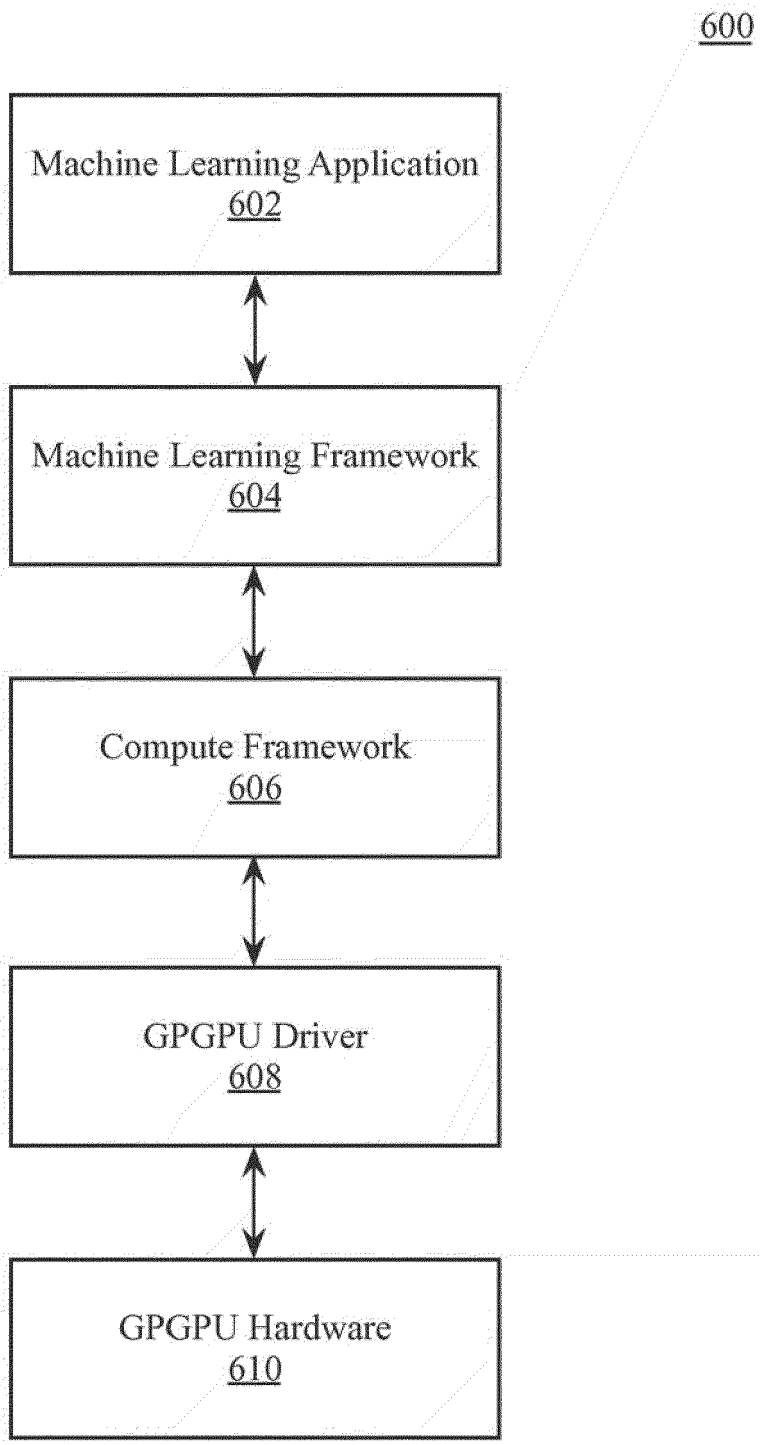

600
Machine Learning Application
602
Machine Learning Framework
604
Compute Framework
606
GPGPU Driver
608
GPGPU Hardware
610

FIG. 6

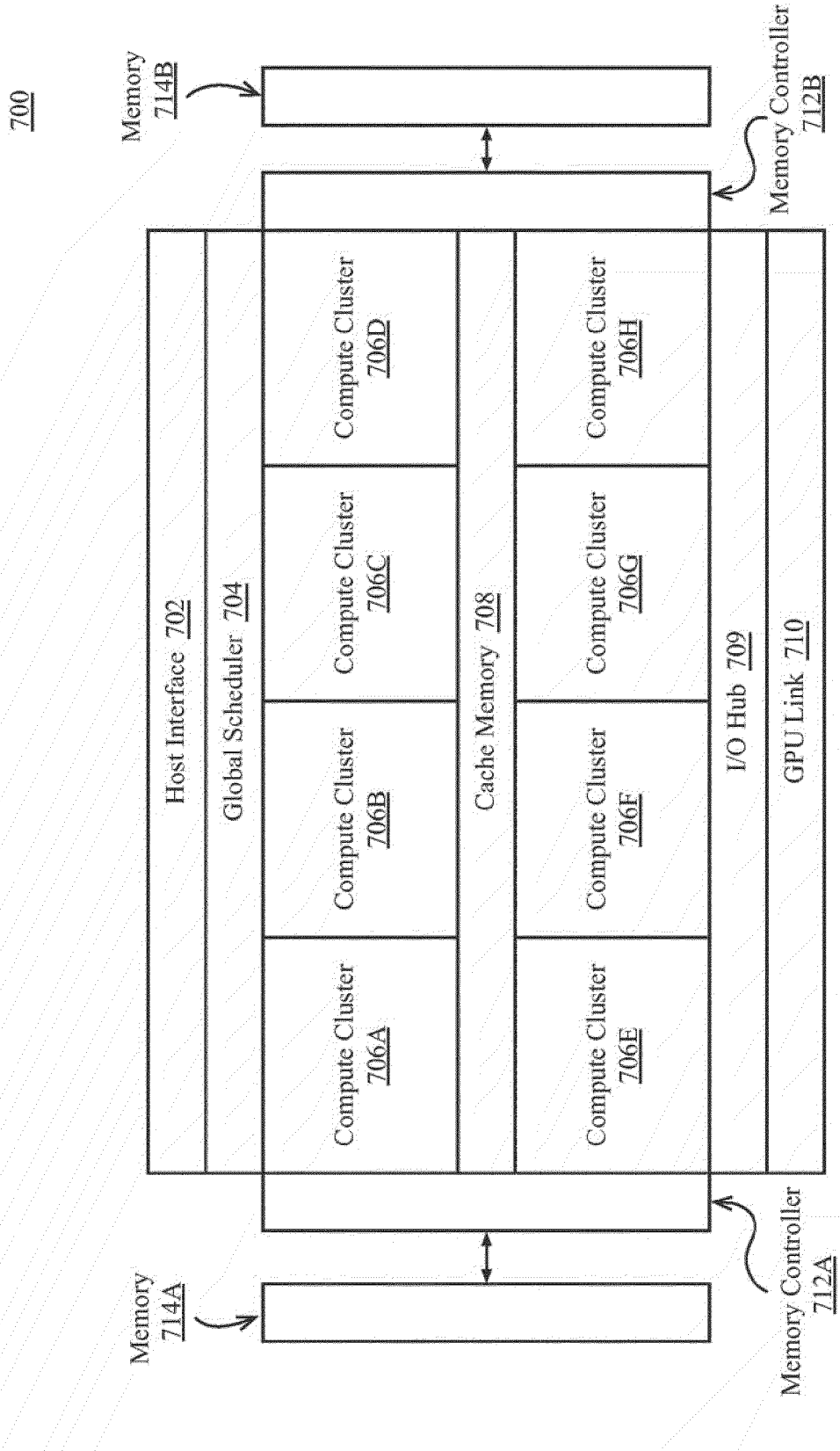

700
Memory 714B
Memory Controller 712B
Host Interface 702
Global Scheduler 704
Compute Cluster 706A
Compute Cluster 706B
Compute Cluster 706C
Compute Cluster 706D
Cache Memory 708
Compute Cluster 706E
Compute Cluster 706F
Compute Cluster 706G
Compute Cluster 706H
I/O Hub 709
GPU Link 710
Memory 714A
Memory Controller 712A

FIG. 7

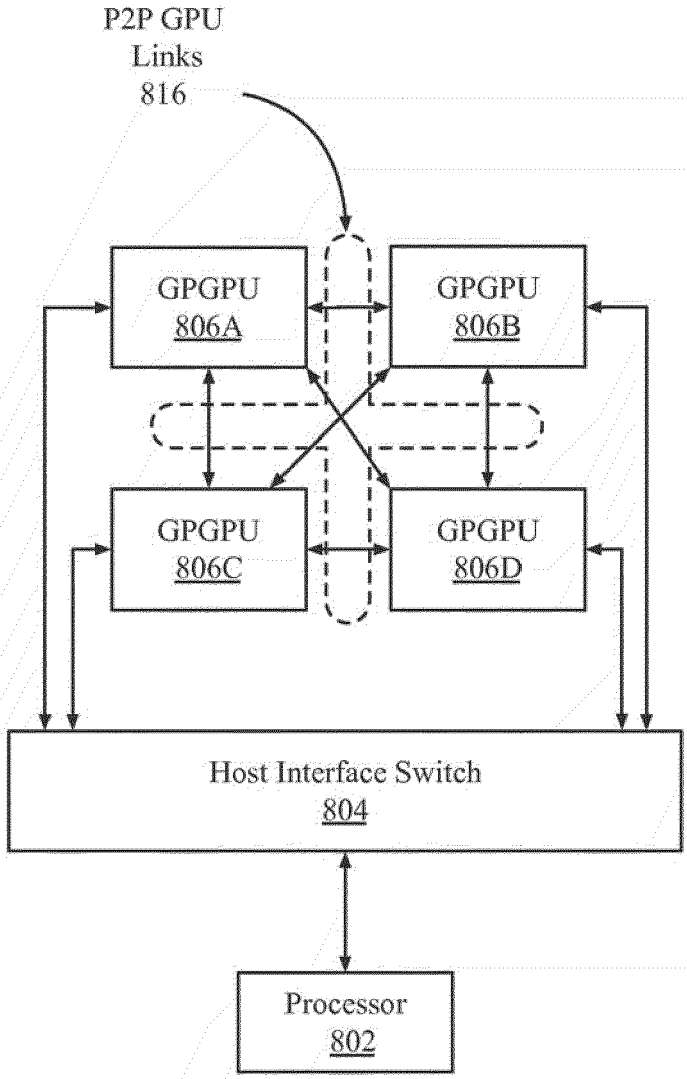

P2P GPU
Links
816
GPGPU
806A
GPGPU
806B
GPGPU
806C
GPGPU
806D
Host Interface Switch
804
Processor
802

FIG. 8

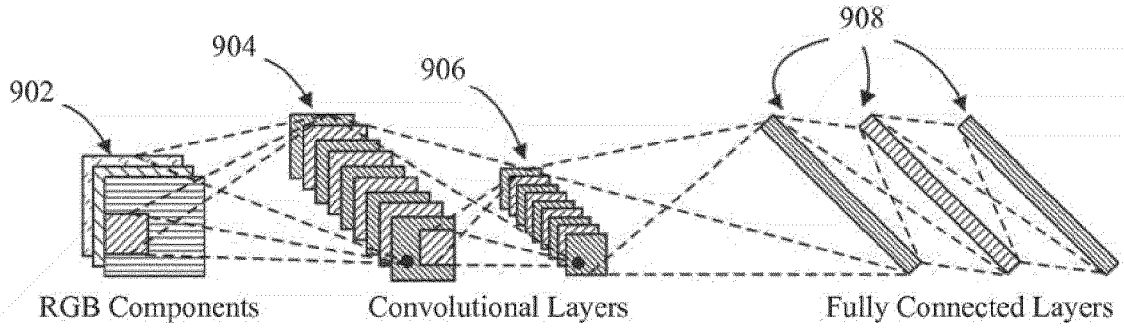

902
904
906
908
RGB Components
Convolutional Layers
Fully Connected Layers

FIG. 9A

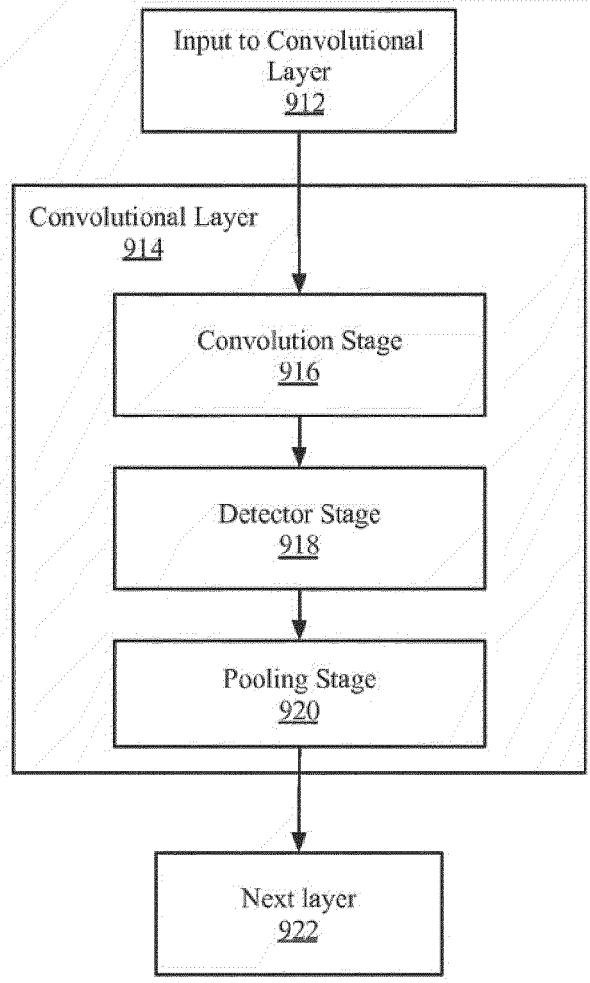

Input to Convolutional Layer
912
Convolutional Layer
914
Convolution Stage
916
Detector Stage
918
Pooling Stage
920
Next layer
922

FIG. 9B

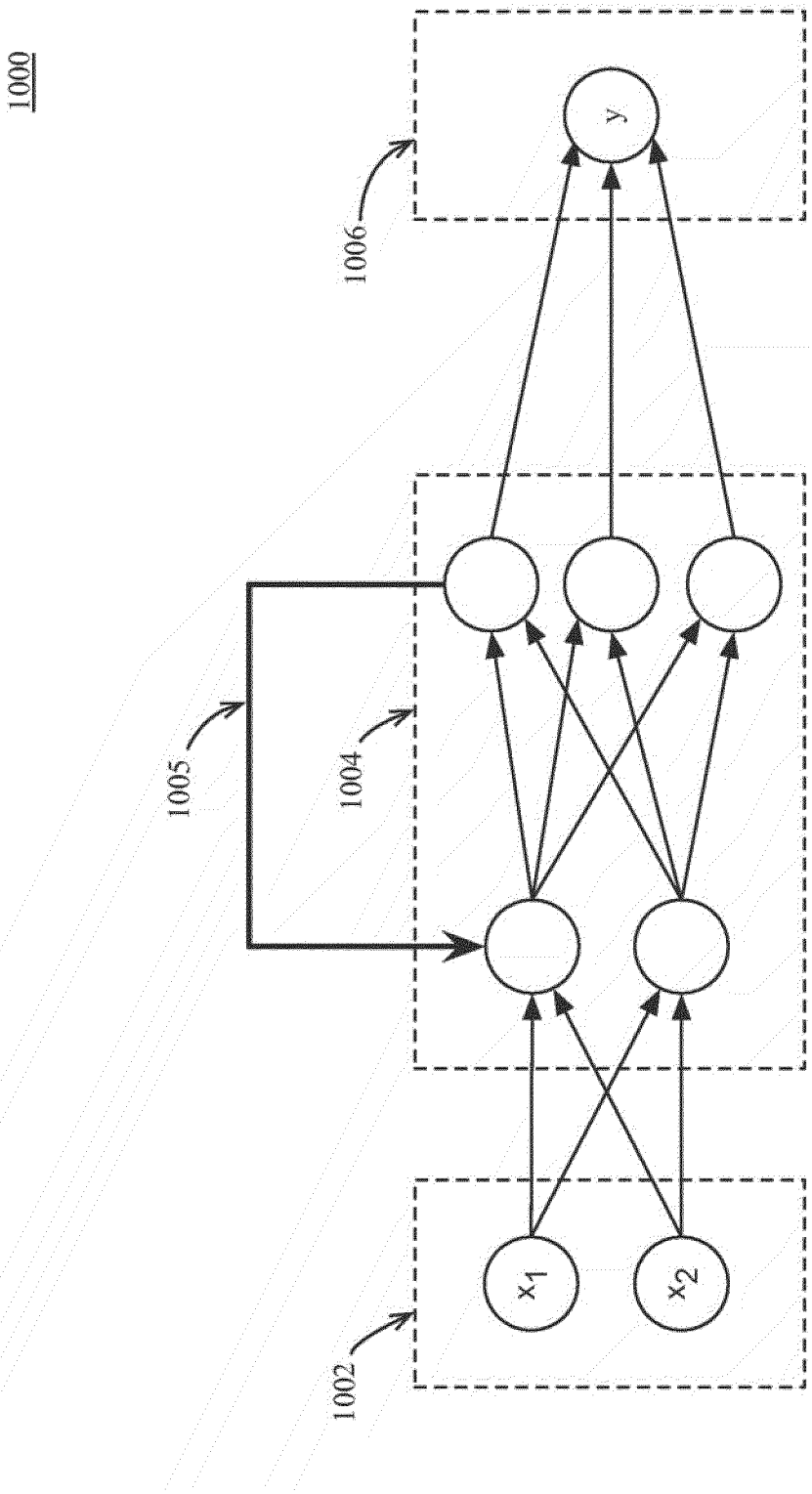


FIG. 10

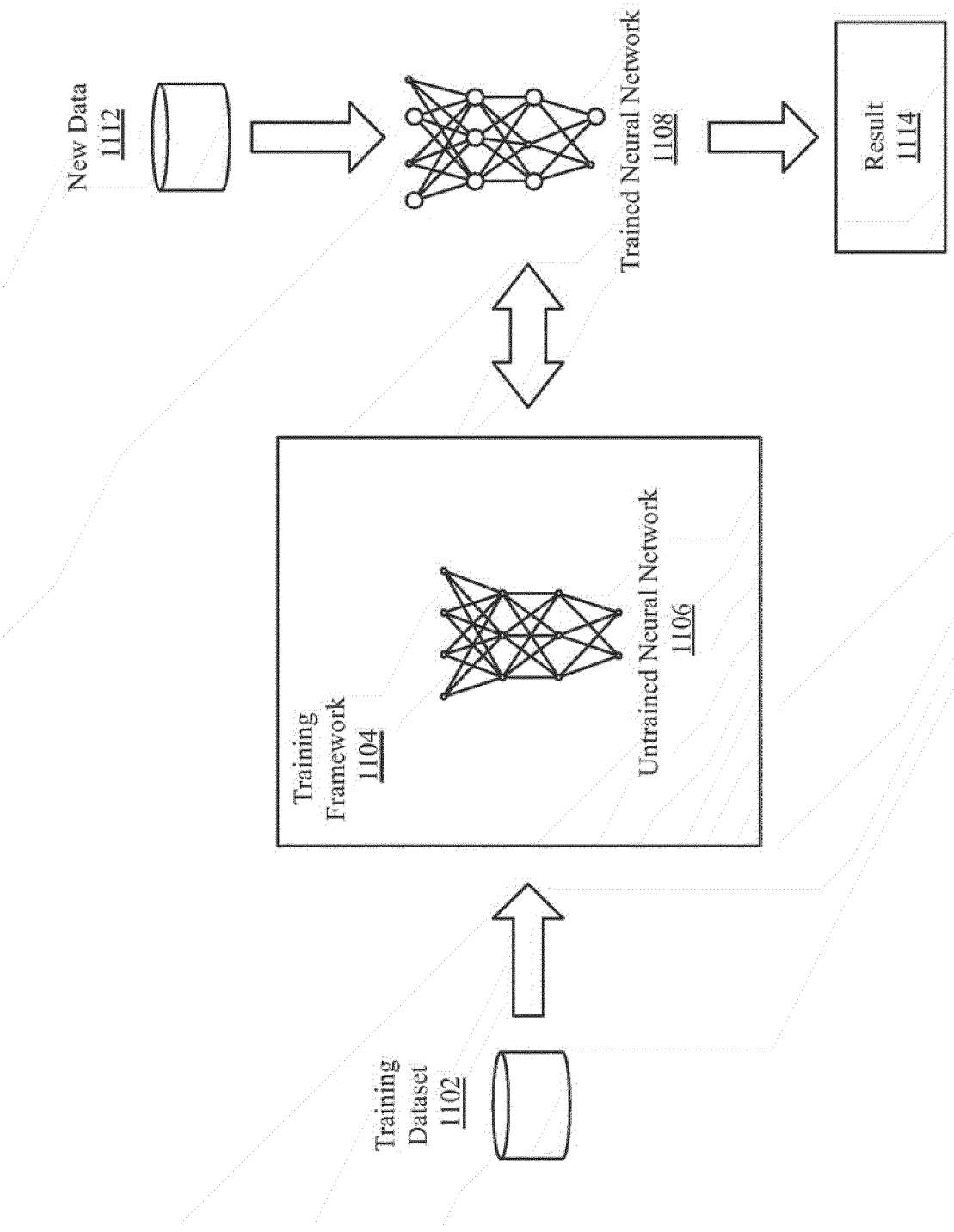

New Data
1112
Trained Neural Network
1108
Result
1114
Training
Framework
1104
Untrained Neural Network
1106
Training
Dataset
1102

FIG. 11

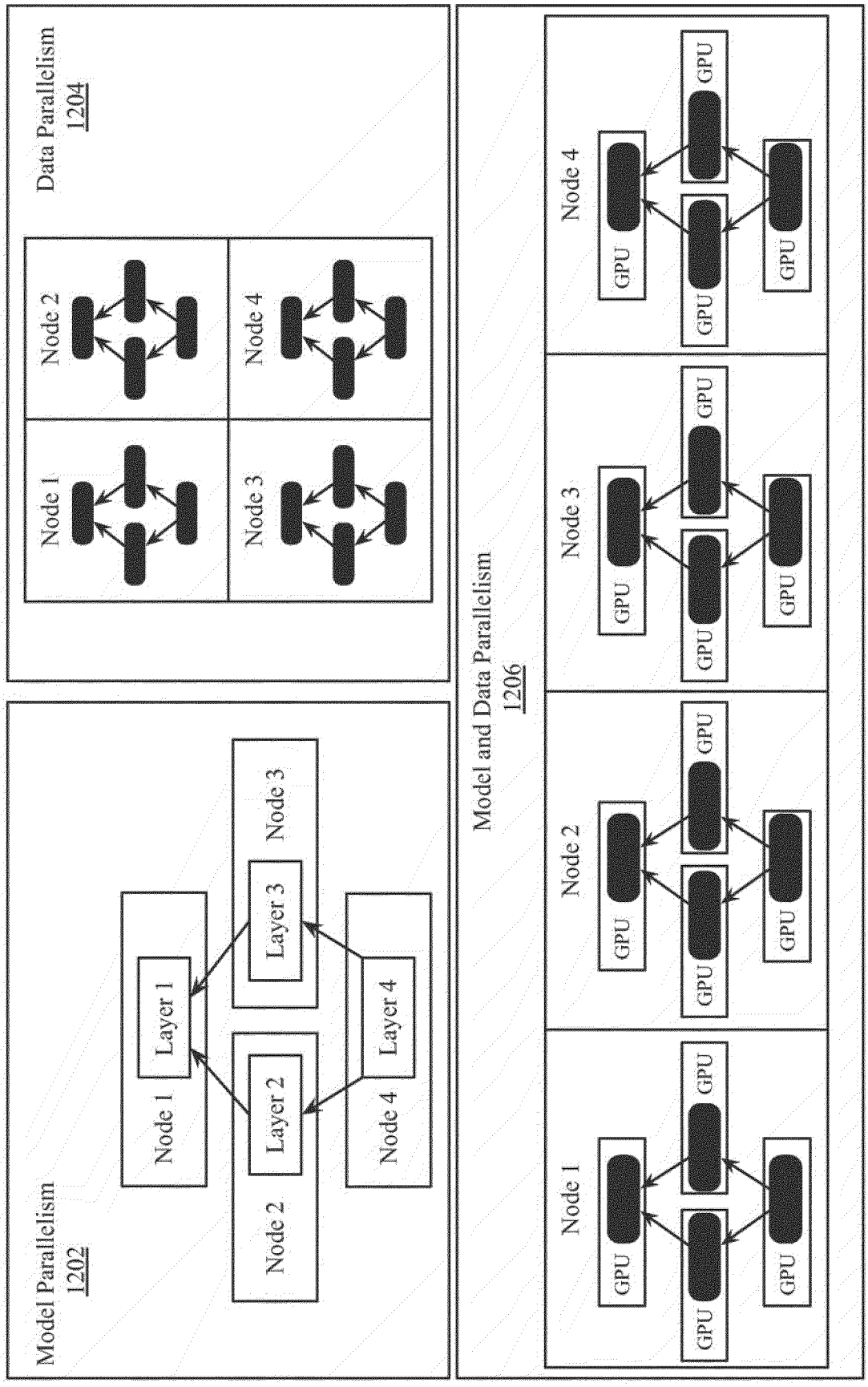

Model Parallelism
1202
Node 1
Layer 1
Node 2
Layer 2
Node 3
Layer 3
Node 4
Layer 4
Data Parallelism
1204
Node 1
Node 2
Node 3
Node 4
Model and Data Parallelism
1206
Node 1
Node 2
Node 3
Node 4
GPU

FIG. 12

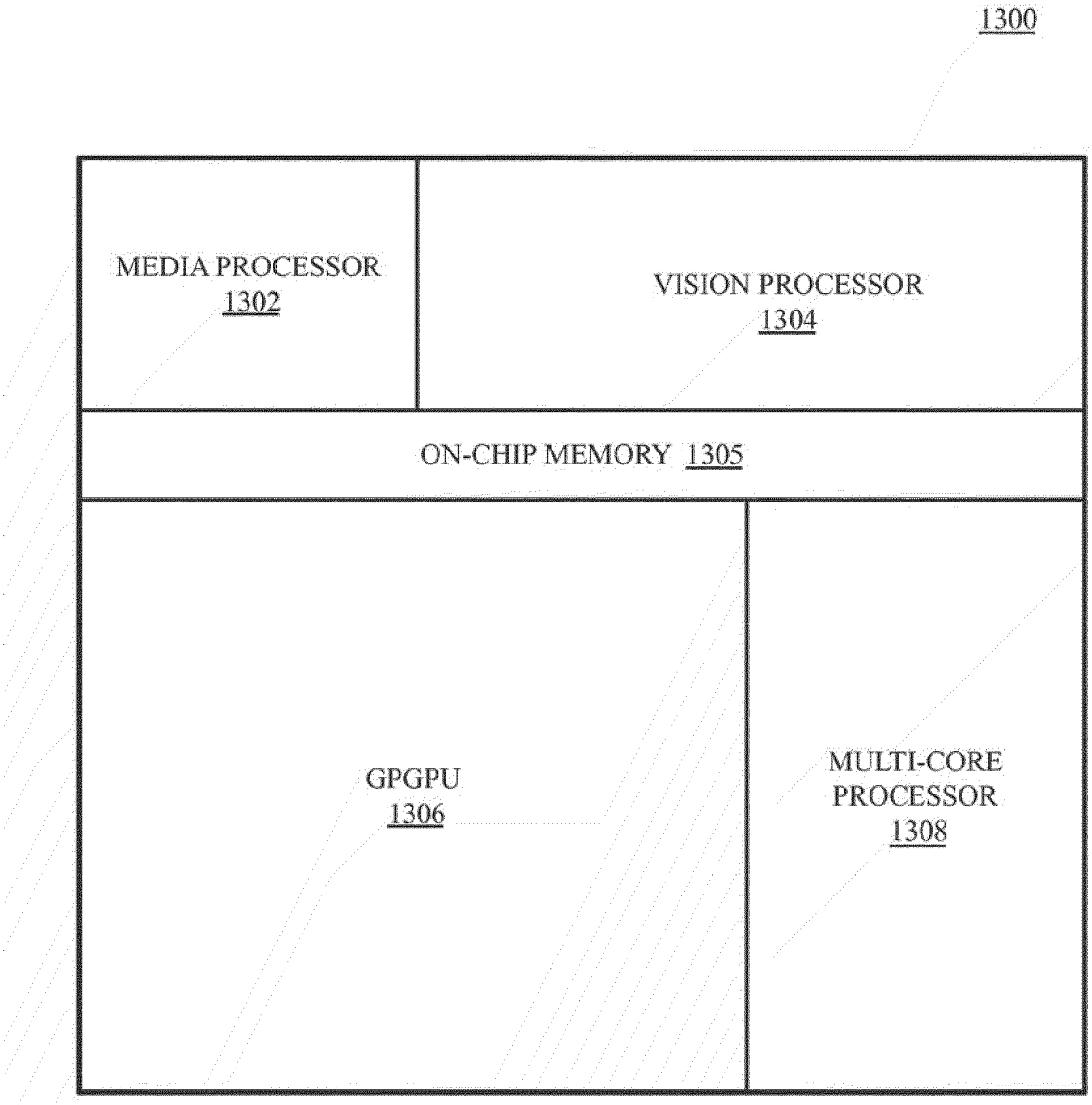

1300
MEDIA PROCESSOR
1302
VISION PROCESSOR
1304
ON-CHIP MEMORY 1305
GPGPU
1306
MULTI-CORE
PROCESSOR
1308

FIG. 13

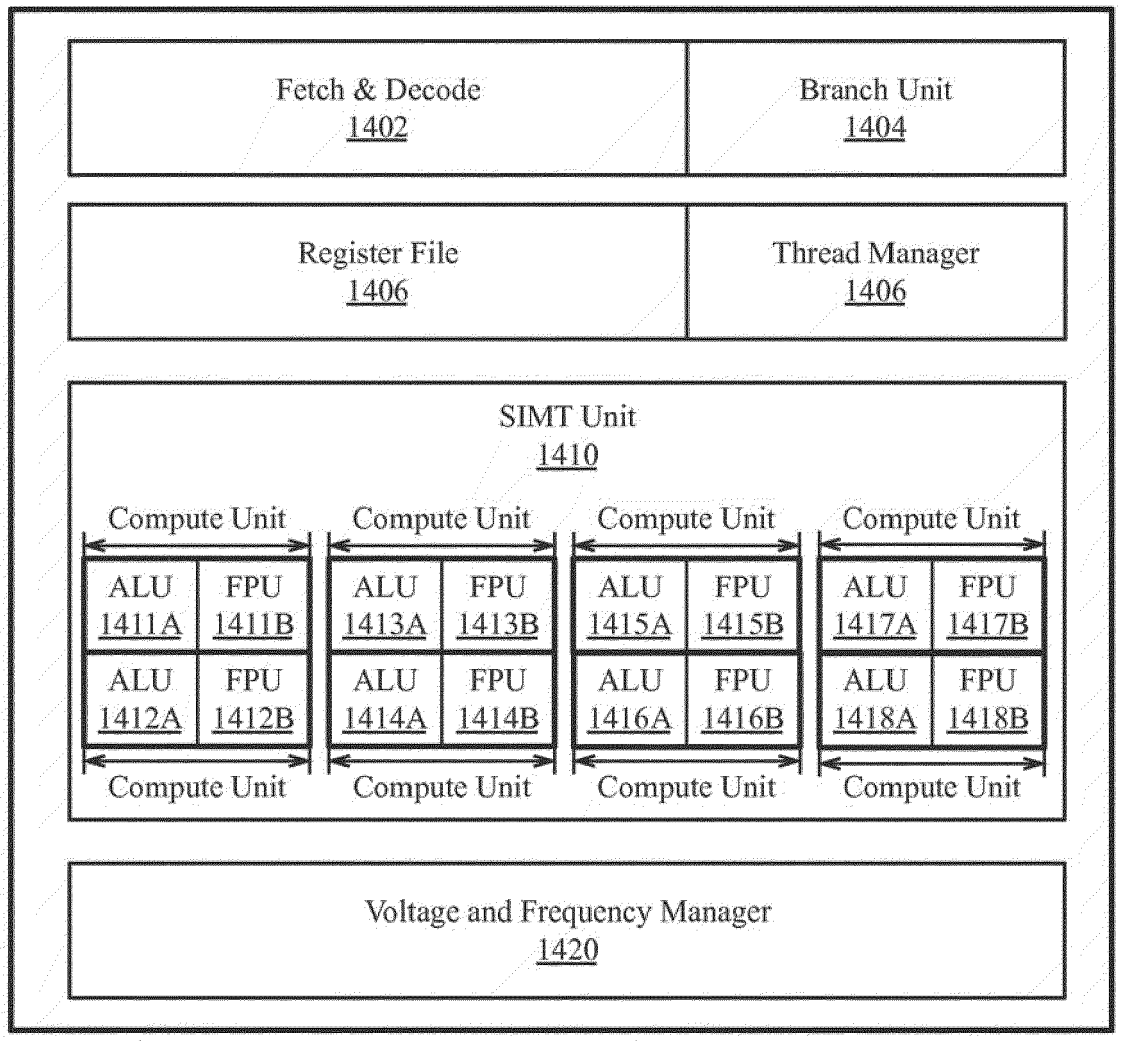

1400
Fetch & Decode
1402
Branch Unit
1404
Register File
1406
Thread Manager
1406
SIMT Unit
1410
Compute Unit
Compute Unit
Compute Unit
Compute Unit
ALU
1411A
FPU
1411B
ALU
1413A
FPU
1413B
ALU
1415A
FPU
1415B
ALU
1417A
FPU
1417B
ALU
1412A
FPU
1412B
ALU
1414A
FPU
1414B
ALU
1416A
FPU
1416B
ALU
1418A
FPU
1418B
Compute Unit
Compute Unit
Compute Unit
Compute Unit
Voltage and Frequency Manager
1420

FIG. 14

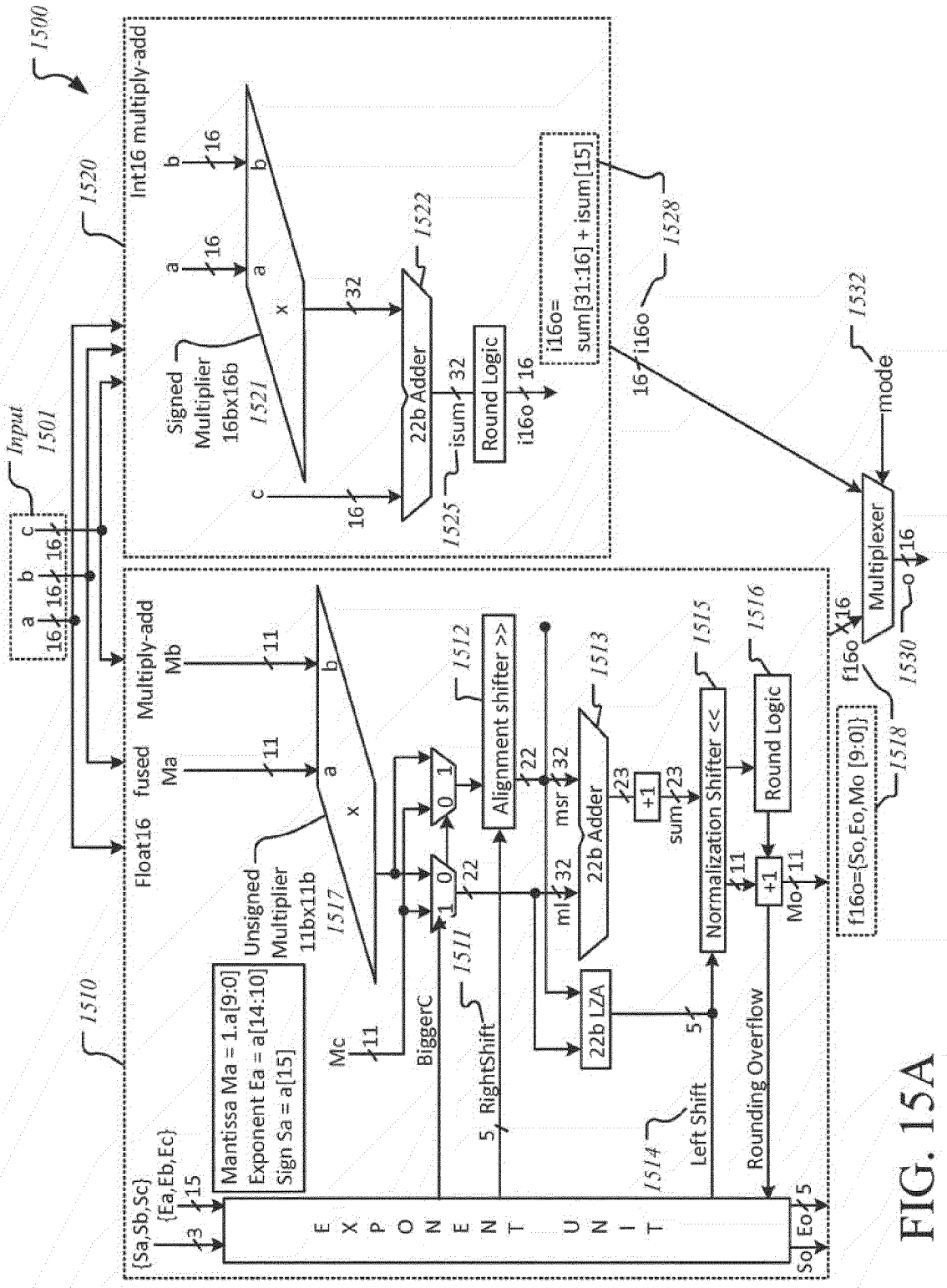

1500
Input
1501
1520
Int16 multiply-add
Signed Multiplier 16bx16b
1521
1522
22b Adder
Round Logic
i16o
isum
1525
i16o= sum[31:16] + isum[15]
1528
1532
mode
Multiplexer
1530
f16o
f16o={So,Eo,Mo [9:0]}
1518
1510
Float16 fused Multiply-add
Unsigned Multiplier 11bx11b
1517
Mantissa Ma = 1.a[9:0]
Exponent Ea = a[14:10]
Sign Sa = a[15]
{Sa,Sb,Sc}
{Ea,Eb,Ec}
EXPONENT UNIT
BiggerC
1511
RightShift
1512
Alignment shifter >>
1513
22b Adder
msr
ml
22b LZA
1514
Left Shift
1515
Normalization Shifter <<
1516
Round Logic
Rounding Overflow
So
Eo
Mo

FIG. 15A

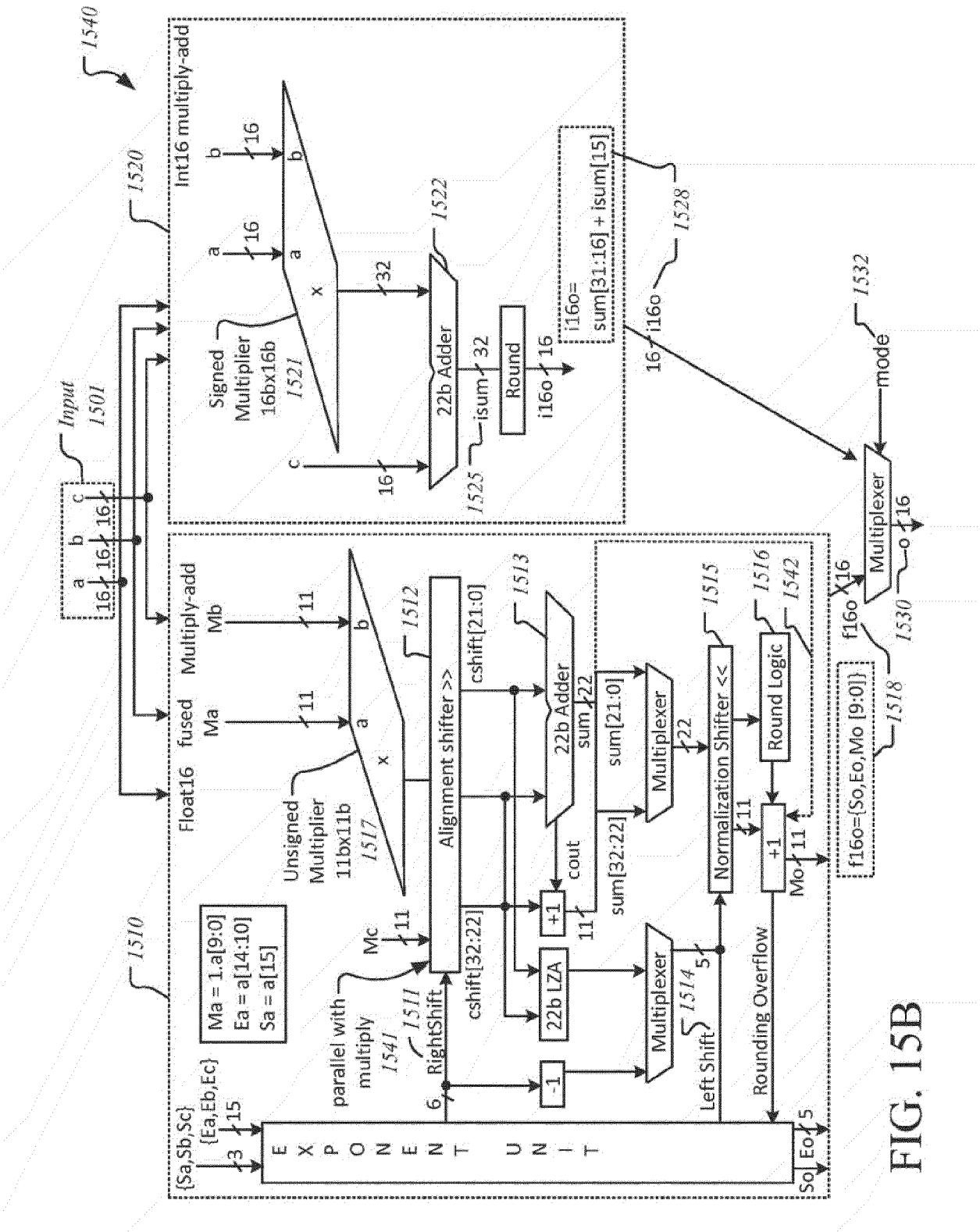

1540
Input
1501
a
b
c
16
1520
Int16 multiply-add
Signed
Multiplier
16bx16b
1521
1522
22b Adder
isum
32
Round
i16o
16
1525
i16o=
sum[31:16] + isum[15]
1528
1532
mode
Multiplexer
o
1530
f16o
1518
f16o={So,Eo,Mo [9:0]}
1510
Float16
fused
Multiply-add
Ma
Mb
11
Unsigned
Multiplier
11bx11b
1517
1512
Alignment shifter >>
cshift[21:0]
cshift[32:22]
1513
22b Adder
sum
22
cout
+1
sum[21:0]
sum[32:22]
Multiplexer
1515
Normalization Shifter <<
1516
Round Logic
1542
Mo
Mc
Ma = 1.a[9:0]
Ea = a[14:10]
Sa = a[15]
parallel with
multiply
1541
1511
RightShift
22b LZA
-1
1514
Left Shift
Rounding Overflow
{Sa,Sb,Sc}
{Ea,Eb,Ec}
3
15
6
5
EXPONENT UNIT
So
Eo

FIG. 15B

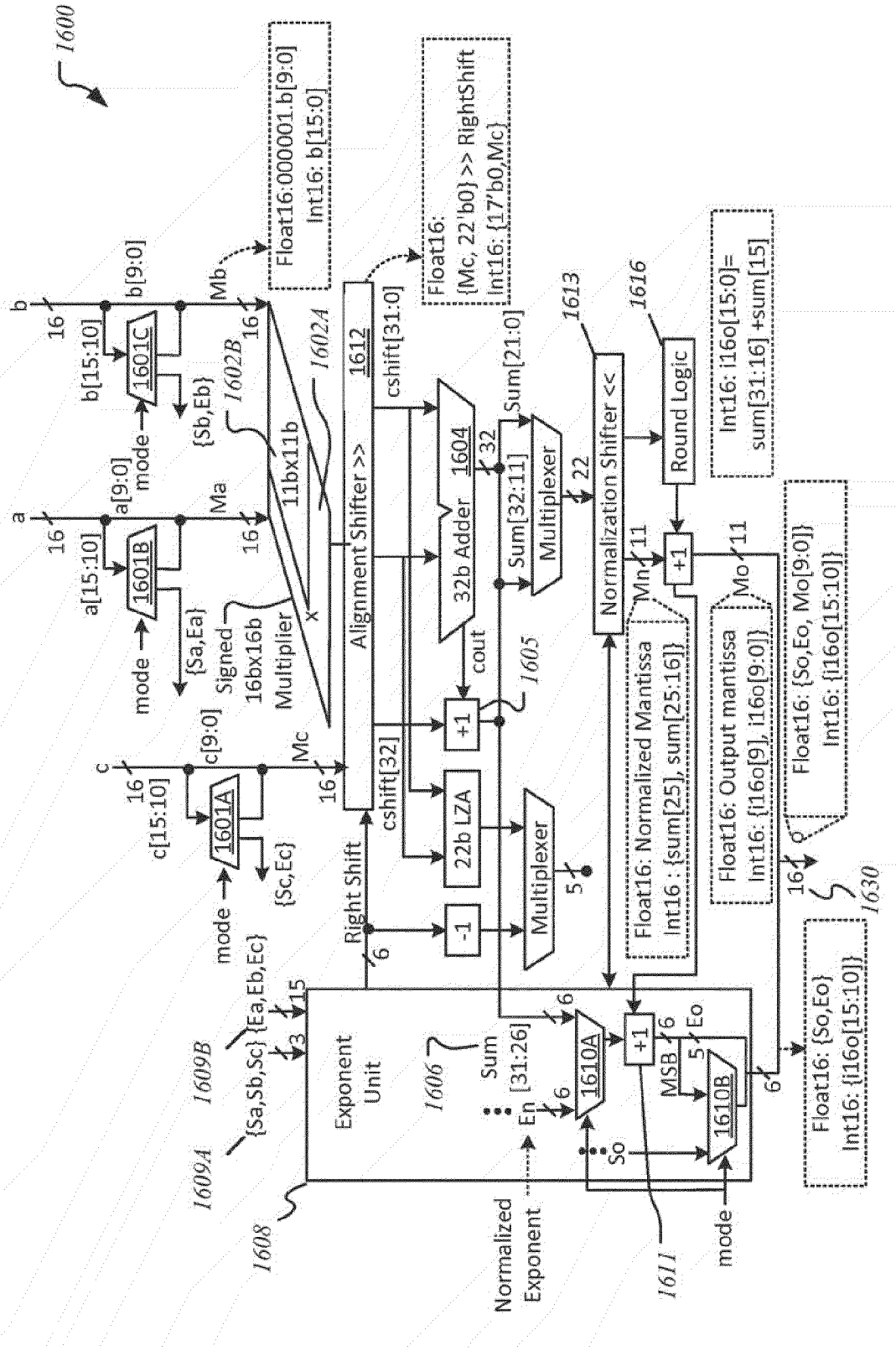

1600
a
16
a[15:10]
a[9:0]
1601B
mode
{Sa,Ea}
Signed
16bx16b
Multiplier
Ma
16
11bx11b
1602A
1602B
b
16
b[15:10]
b[9:0]
1601C
mode
{Sb,Eb}
Mb
Float16:000001.b[9:0]
Int16: b[15:0]
c
16
c[15:10]
c[9:0]
1601A
mode
{Sc,Ec}
Mc
16
Alignment Shifter >>
1612
cshift[32]
cshift[31:0]
Float16:
{Mc, 22'b0} >> RightShift
Int16: {17'b0,Mc}
32b Adder
1604
32
Sum[32:11]
Sum[21:0]
Multiplexer
22
cout
+1
1605
22b LZA
-1
Right Shift
6
Multiplexer
5
1613
Normalization Shifter <<
1616
Round Logic
Mn
11
Float16: Normalized Mantissa
Int16 : {sum[25], sum[25:16]}
+1
Mo
11
Float16: Output mantissa
Int16: {i16o[9], i16o[9:0]}
Int16: i16o[15:0]=
sum[31:16] +sum[15]
1609B
{Sa,Sb,Sc}
{Ea,Eb,Ec}
3
15
1609A
1608
Exponent
Unit
1606
Sum
[31:26]
En
So
6
1610A
+1
MSB
6
Eo
5
1610B
6
1611
mode
Normalized
Exponent
16
Float16: {So,Eo, Mo[9:0]}
Int16: {i16o[15:10]}
Float16: {So,Eo}
Int16: {i16o[15:10]}
1630

FIG. 16

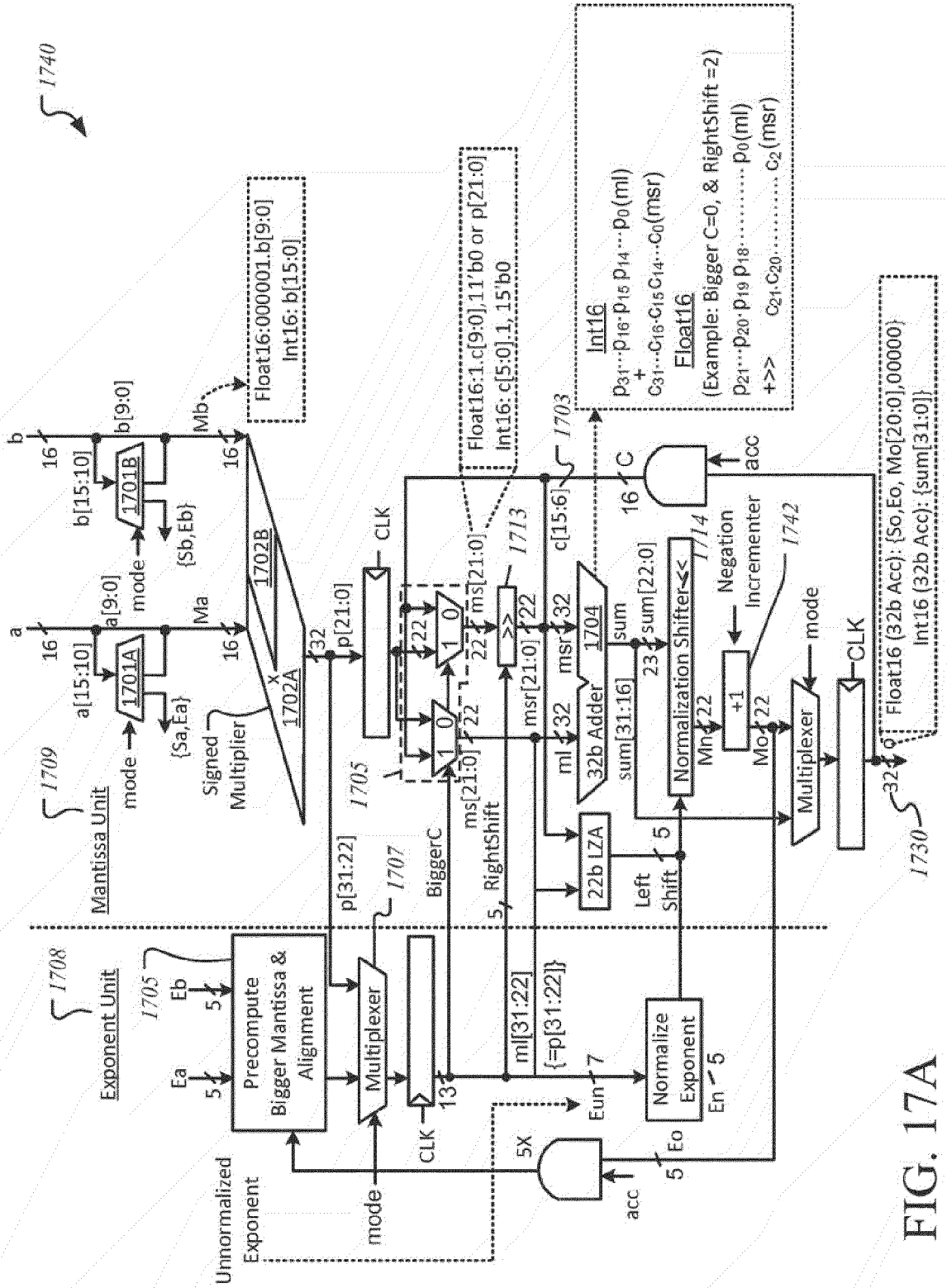

1740
Exponent Unit
1708
Mantissa Unit
1709
Unnormalized Exponent
Precompute Bigger Mantissa & Alignment
1705
Multiplexer
1707
Normalize Exponent
Signed Multiplier
1702A
1702B
1701A
1701B
a[15:10]
a[9:0]
b[15:10]
b[9:0]
{Sa,Ea}
{Sb,Eb}
Float16:000001.b[9:0]
Int16: b[15:0]
Float16:1.c[9:0],11'b0 or p[21:0]
Int16: c[5:0].1, 15'b0
p[31:22]
p[21:0]
BiggerC
RightShift
ml[31:22]
{=p[31:22]}
22b LZA
32b Adder
1704
1713
1703
c[15:6]
Left Shift
Normalization Shifter<<
1714
Negation Incrementer
1742
Multiplexer
1730
Float16 (32b Acc): {So,Eo, Mo[20:0],00000}
Int16 (32b Acc): {sum[31:0]}
Int16
Float16
(Example: Bigger C=0, & RightShift =2)
acc
mode
CLK

FIG. 17A

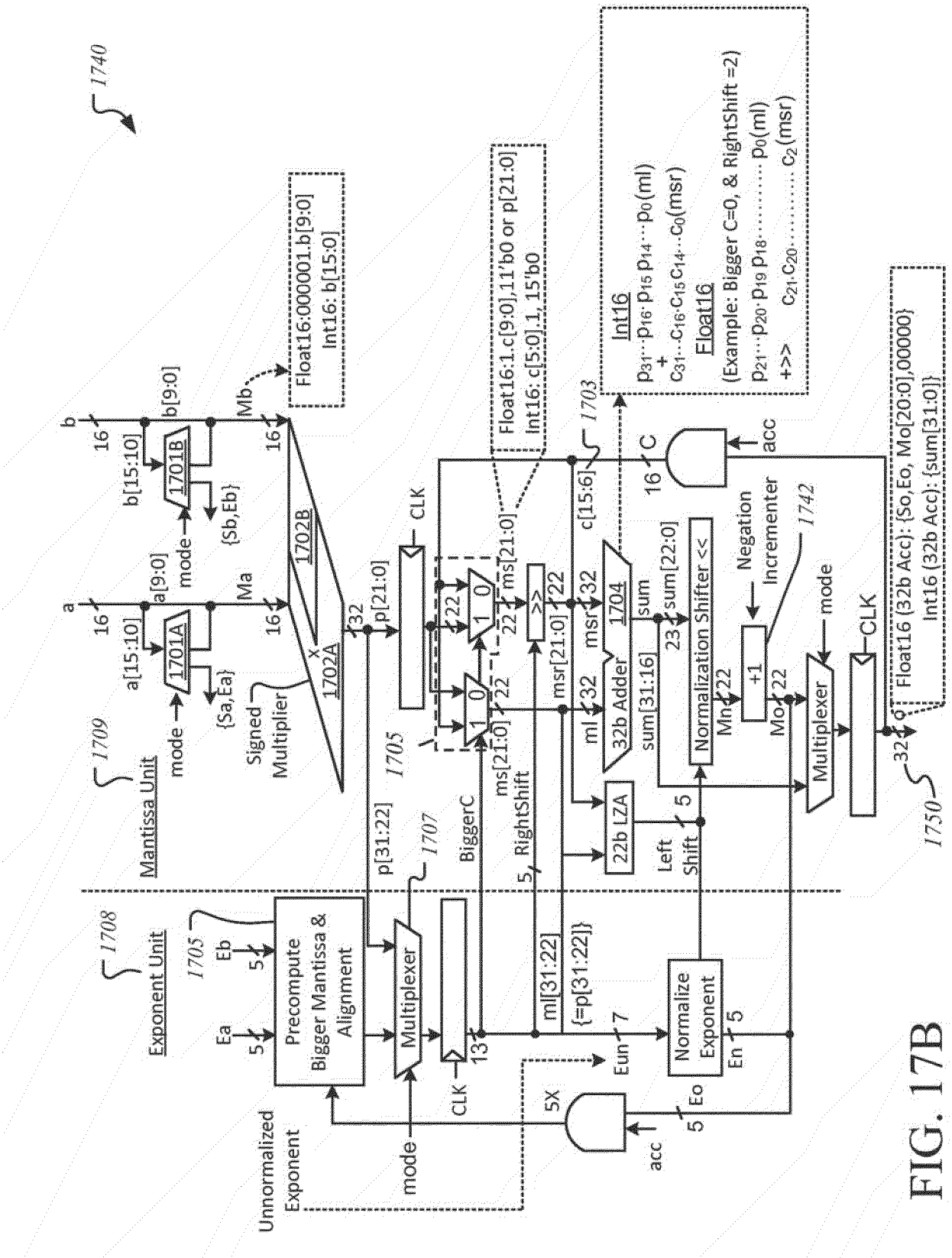

1740
Mantissa Unit
1709
Exponent Unit
1708
Unnormalized Exponent
Precompute Bigger Mantissa & Alignment
1705
Multiplexer
1707
CLK
mode
Ea
Eb
5
13
p[31:22]
BiggerC
RightShift
ml[31:22]
{=p[31:22]}
Eun
7
Normalize Exponent
En
Eo
5X
acc
a
16
a[15:10]
a[9:0]
1701A
{Sa,Ea}
Ma
b
b[15:10]
b[9:0]
1701B
{Sb,Eb}
Mb
Signed Multiplier
1702A
1702B
32
p[21:0]
Float16:000001.b[9:0]
Int16: b[15:0]
Float16:1.c[9:0],11'b0 or p[21:0]
Int16: c[5:0].1, 15'b0
ms[21:0]
22
1 0
ms[21:0]
msr[21:0]
>>
ml
msr
32b Adder
1704
sum
sum[31:16]
sum[22:0]
23
22b LZA
Left Shift
c[15:6]
1703
C
Normalization Shifter <<
Mn
Mo
+1
Negation Incrementer
1742
Multiplexer
Float16 (32b Acc): {So,Eo, Mo[20:0],00000}
Int16 (32b Acc): {sum[31:0]}
1750
Int16
$p_{31}\cdots p_{16}\cdot p_{15}\, p_{14}\cdots p_0$(ml)
+
$c_{31}\cdots c_{16}\cdot c_{15}\, c_{14}\cdots c_0$(msr)
Float16
(Example: Bigger C=0, & RightShift =2)
$p_{21}\cdots p_{20}\cdot p_{19}\, p_{18}\cdots\cdots p_0$(ml)
+>> $c_{21}\cdot c_{20}\cdots\cdots c_2$(msr)

FIG. 17B

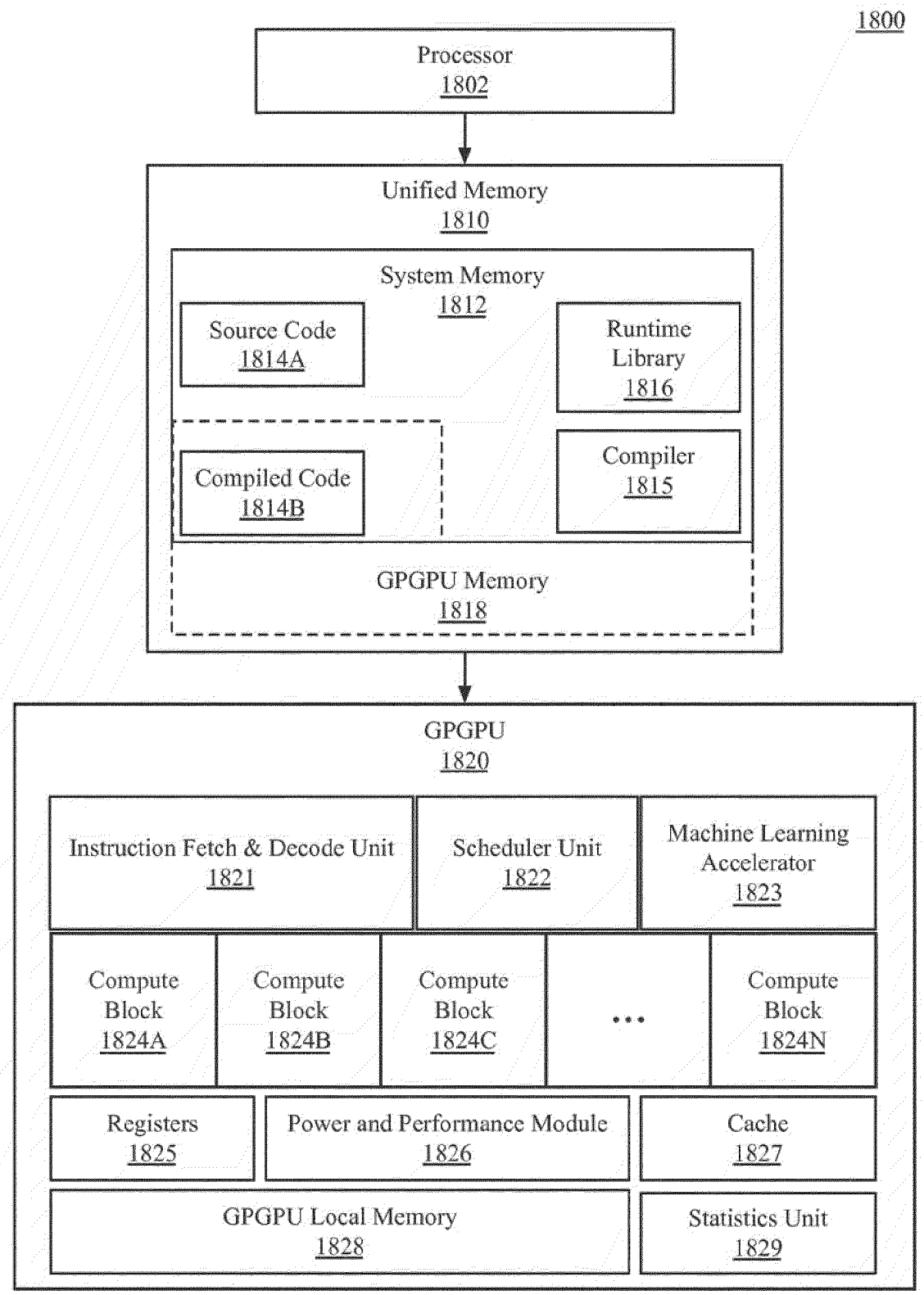

1800
Processor
1802
Unified Memory
1810
System Memory
1812
Source Code
1814A
Runtime
Library
1816
Compiled Code
1814B
Compiler
1815
GPGPU Memory
1818
GPGPU
1820
Instruction Fetch & Decode Unit
1821
Scheduler Unit
1822
Machine Learning
Accelerator
1823
Compute
Block
1824A
Compute
Block
1824B
Compute
Block
1824C
...
Compute
Block
1824N
Registers
1825
Power and Performance Module
1826
Cache
1827
GPGPU Local Memory
1828
Statistics Unit
1829

FIG. 18A

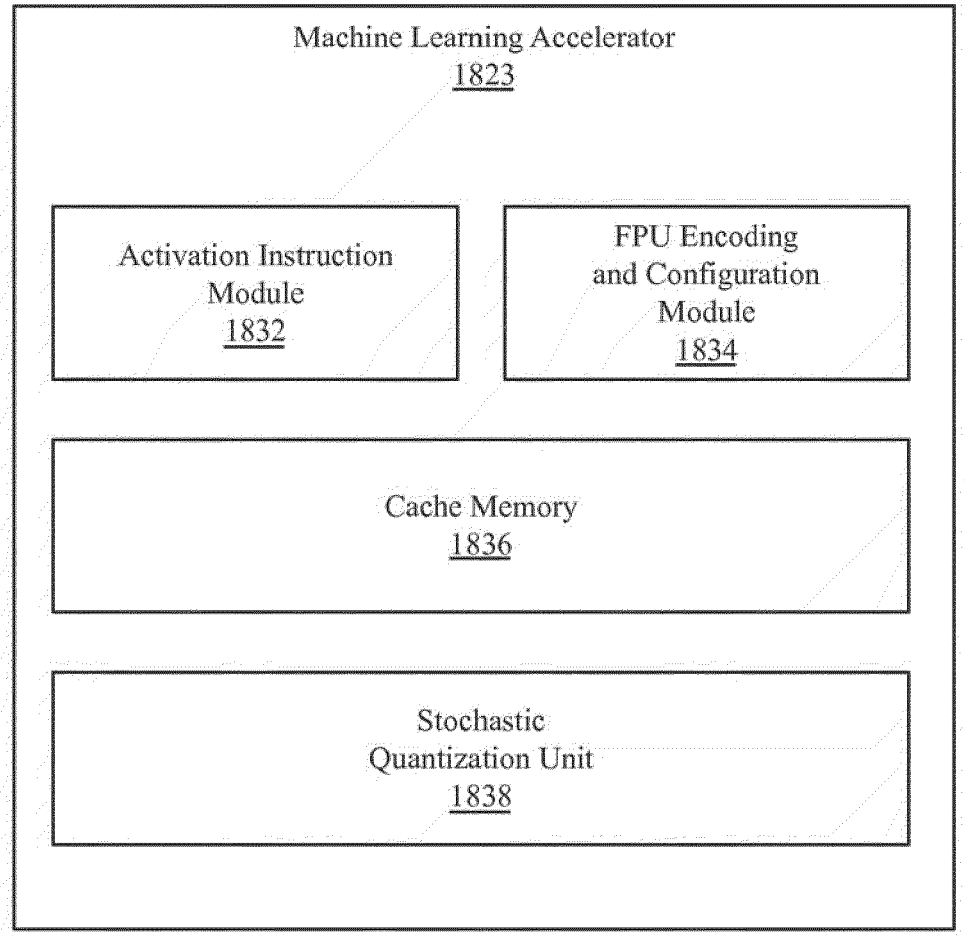

Machine Learning Accelerator
1823
Activation Instruction Module
1832
FPU Encoding and Configuration Module
1834
Cache Memory
1836
Stochastic Quantization Unit
1838

FIG. 18B

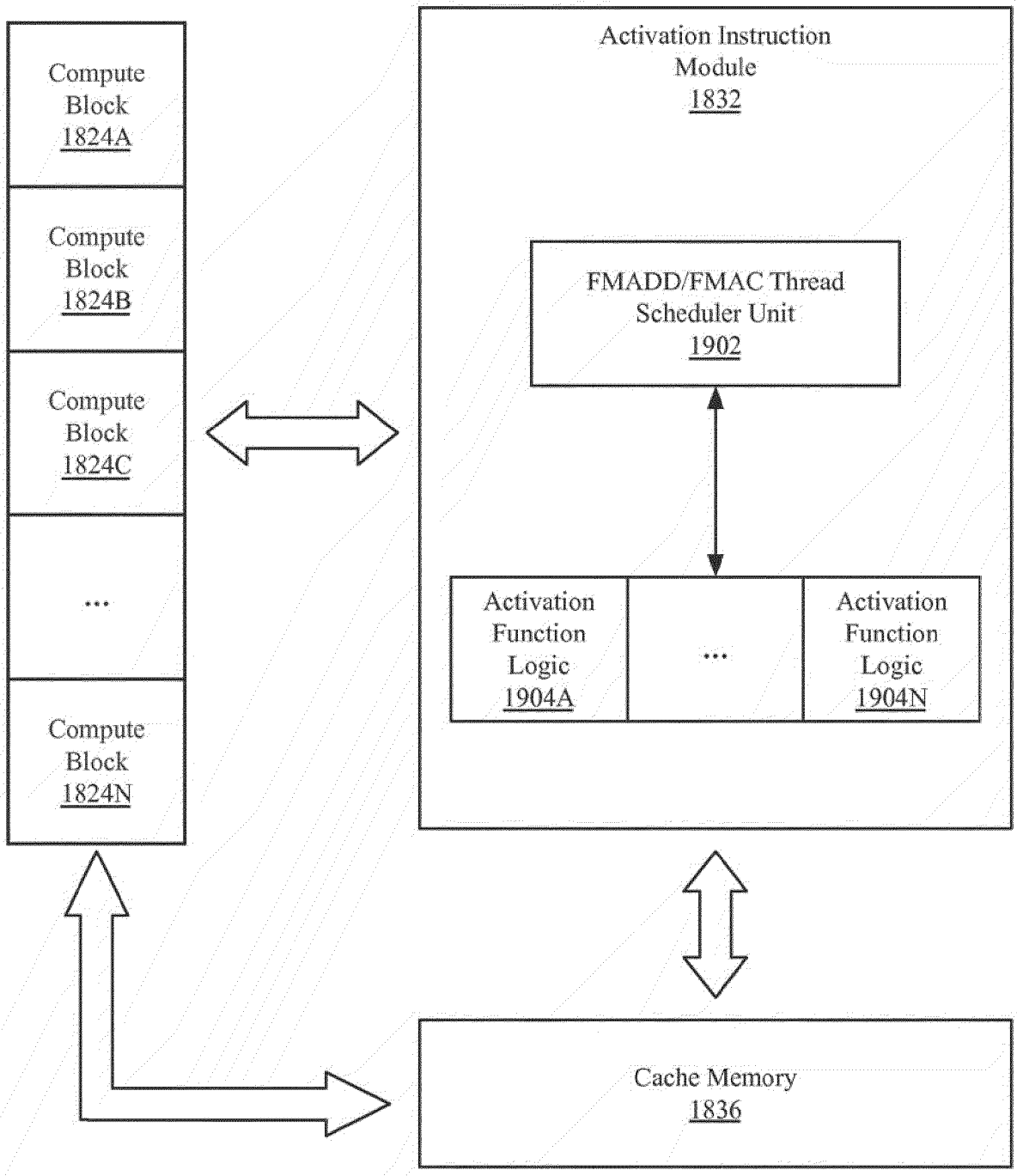

Compute Block 1824A
Compute Block 1824B
Compute Block 1824C
...
Compute Block 1824N
Activation Instruction Module 1832
FMADD/FMAC Thread Scheduler Unit 1902
Activation Function Logic 1904A
...
Activation Function Logic 1904N
Cache Memory 1836

FIG. 19

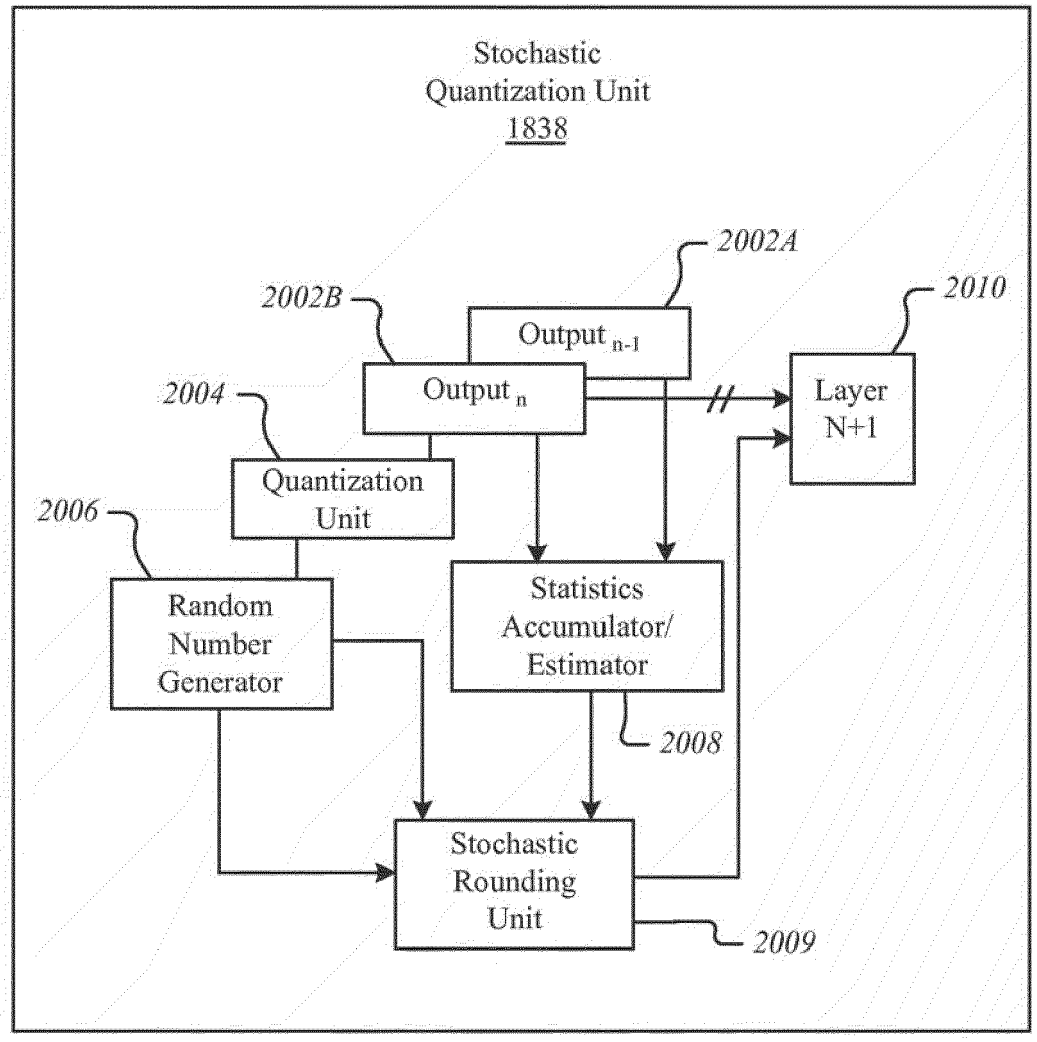

Stochastic
Quantization Unit
1838
2002A
2002B
2010
Output n-1
Output n
Layer
N+1
2004
Quantization
Unit
2006
Random
Number
Generator
Statistics
Accumulator/
Estimator
2008
Stochastic
Rounding
Unit
2009

FIG. 20

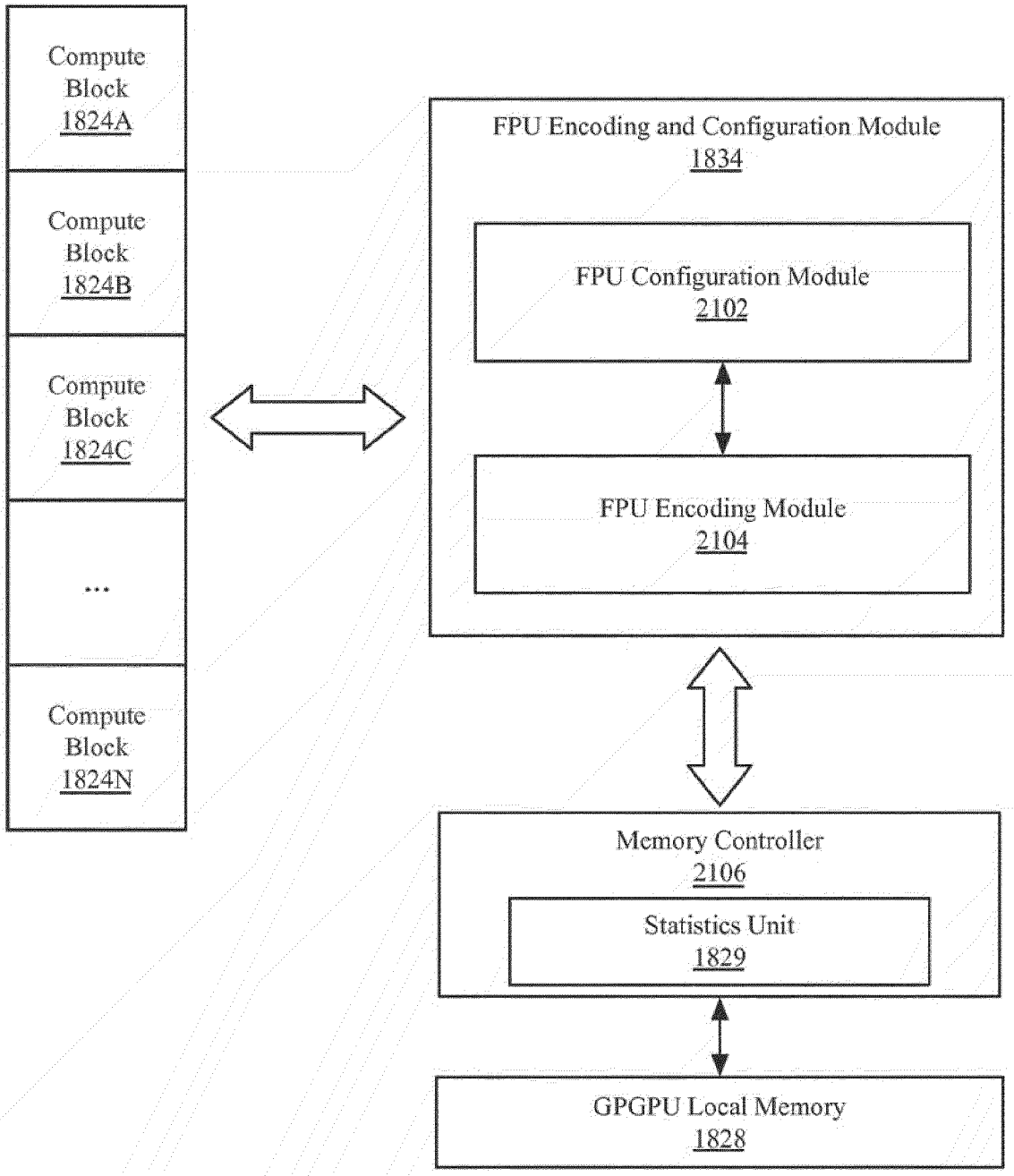

Compute Block 1824A
Compute Block 1824B
Compute Block 1824C
...
Compute Block 1824N
FPU Encoding and Configuration Module 1834
FPU Configuration Module 2102
FPU Encoding Module 2104
Memory Controller 2106
Statistics Unit 1829
GPGPU Local Memory 1828

FIG. 21

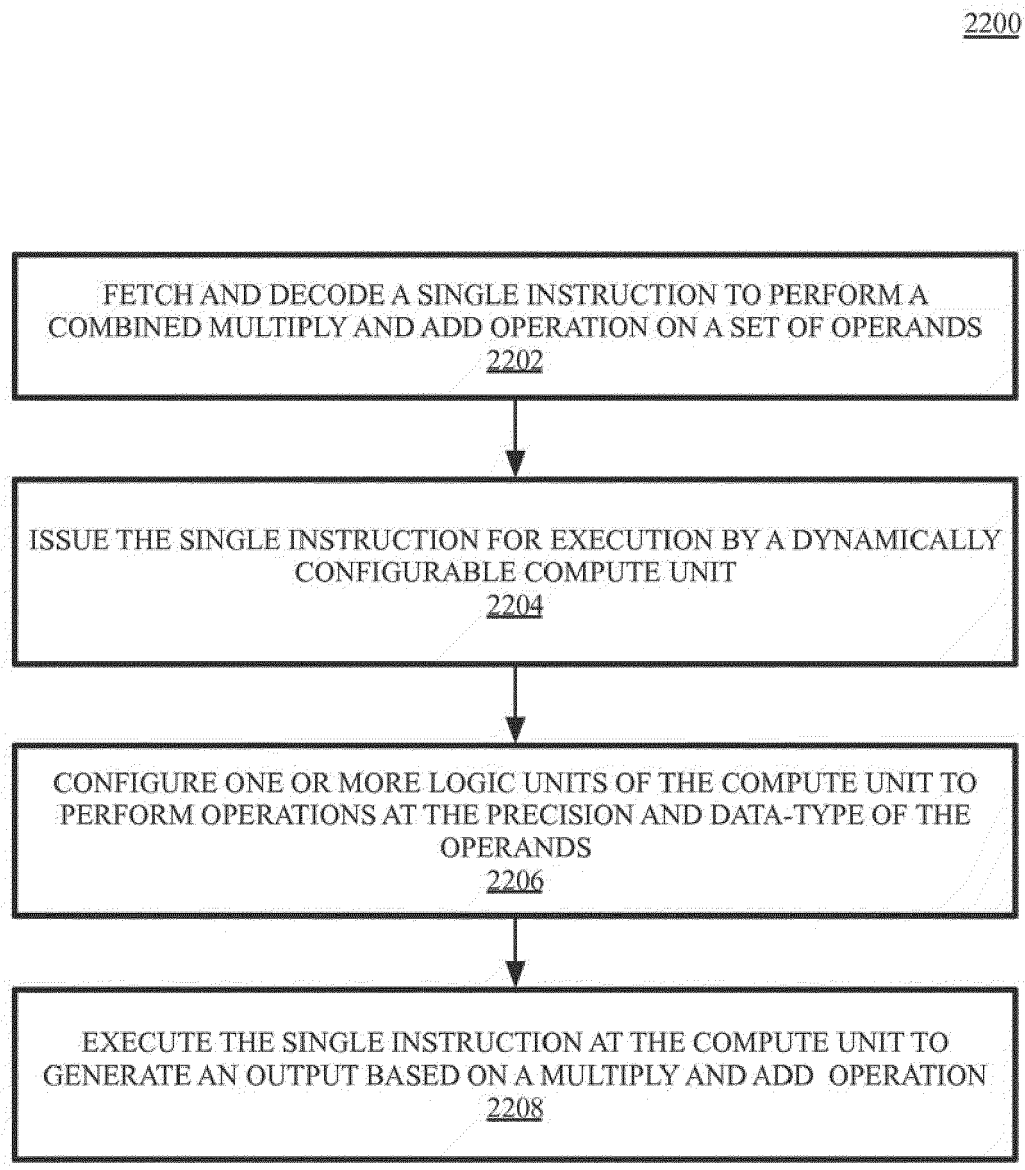

2200
FETCH AND DECODE A SINGLE INSTRUCTION TO PERFORM A COMBINED MULTIPLY AND ADD OPERATION ON A SET OF OPERANDS
2202
ISSUE THE SINGLE INSTRUCTION FOR EXECUTION BY A DYNAMICALLY CONFIGURABLE COMPUTE UNIT
2204
CONFIGURE ONE OR MORE LOGIC UNITS OF THE COMPUTE UNIT TO PERFORM OPERATIONS AT THE PRECISION AND DATA-TYPE OF THE OPERANDS
2206
EXECUTE THE SINGLE INSTRUCTION AT THE COMPUTE UNIT TO GENERATE AN OUTPUT BASED ON A MULTIPLY AND ADD OPERATION
2208

FIG. 22

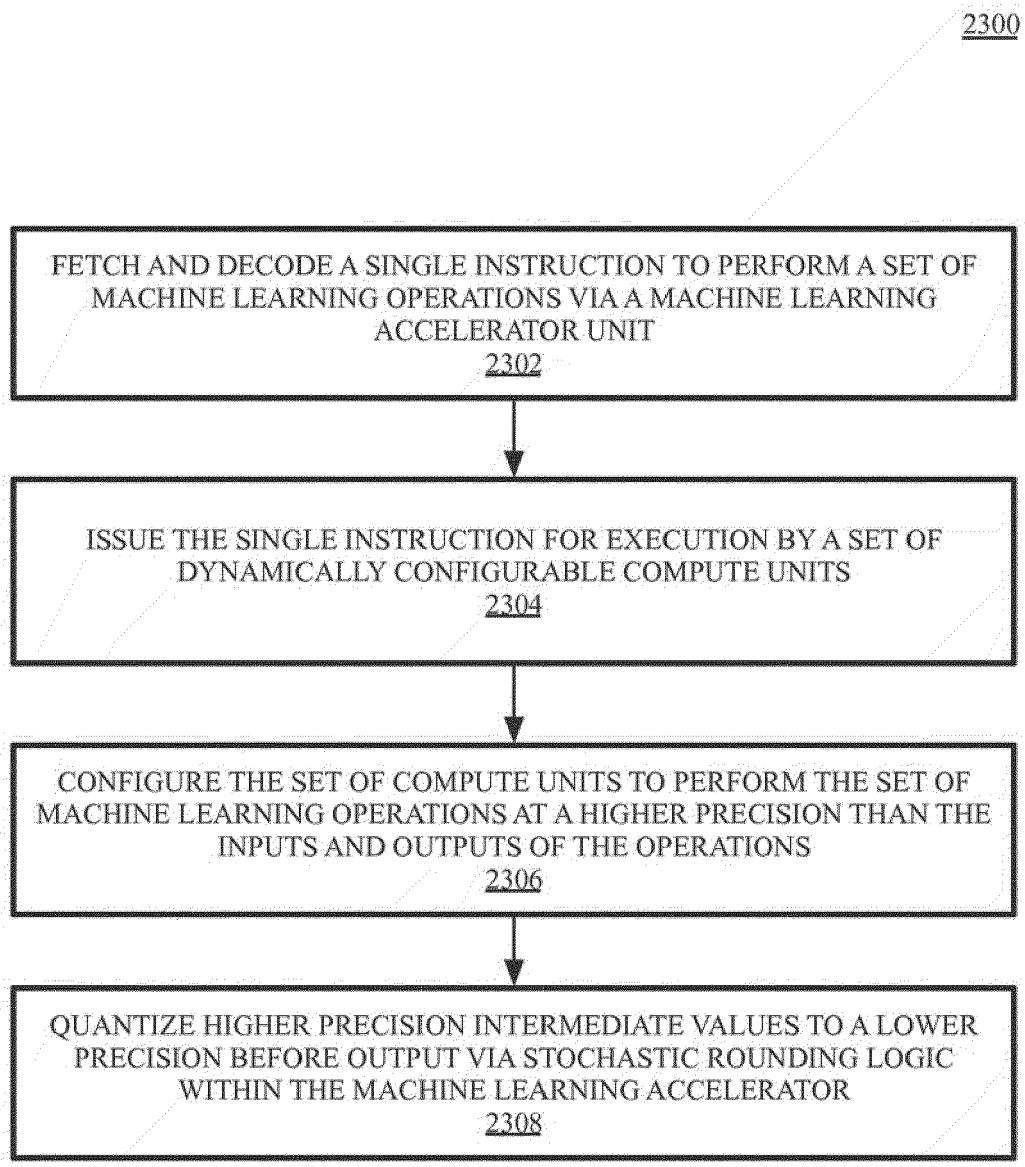

2300
FETCH AND DECODE A SINGLE INSTRUCTION TO PERFORM A SET OF MACHINE LEARNING OPERATIONS VIA A MACHINE LEARNING ACCELERATOR UNIT
2302
ISSUE THE SINGLE INSTRUCTION FOR EXECUTION BY A SET OF DYNAMICALLY CONFIGURABLE COMPUTE UNITS
2304
CONFIGURE THE SET OF COMPUTE UNITS TO PERFORM THE SET OF MACHINE LEARNING OPERATIONS AT A HIGHER PRECISION THAN THE INPUTS AND OUTPUTS OF THE OPERATIONS
2306
QUANTIZE HIGHER PRECISION INTERMEDIATE VALUES TO A LOWER PRECISION BEFORE OUTPUT VIA STOCHASTIC ROUNDING LOGIC WITHIN THE MACHINE LEARNING ACCELERATOR
2308

FIG. 23A

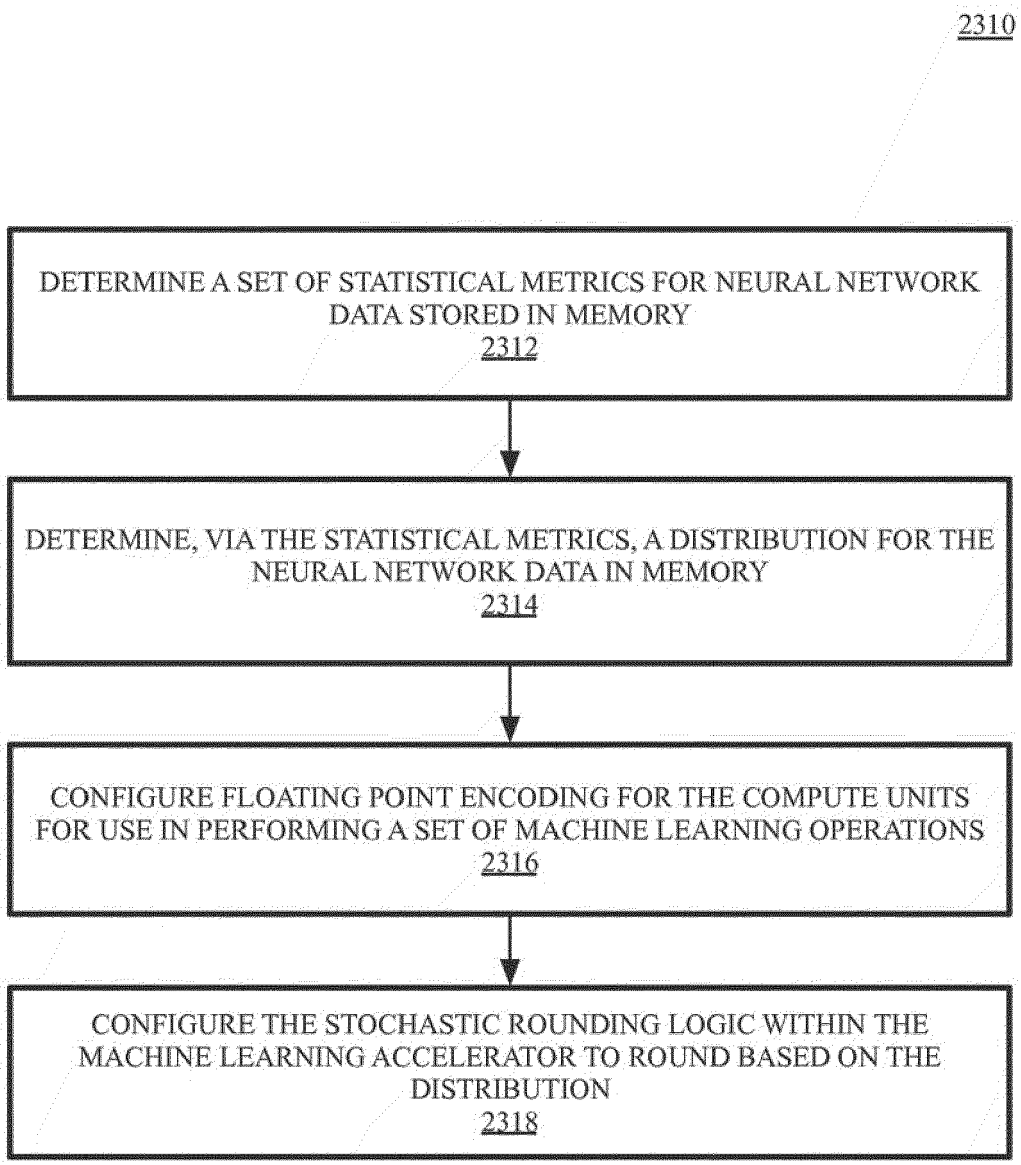

2310
DETERMINE A SET OF STATISTICAL METRICS FOR NEURAL NETWORK DATA STORED IN MEMORY
2312
DETERMINE, VIA THE STATISTICAL METRICS, A DISTRIBUTION FOR THE NEURAL NETWORK DATA IN MEMORY
2314
CONFIGURE FLOATING POINT ENCODING FOR THE COMPUTE UNITS FOR USE IN PERFORMING A SET OF MACHINE LEARNING OPERATIONS
2316
CONFIGURE THE STOCHASTIC ROUNDING LOGIC WITHIN THE MACHINE LEARNING ACCELERATOR TO ROUND BASED ON THE DISTRIBUTION
2318

FIG. 23B

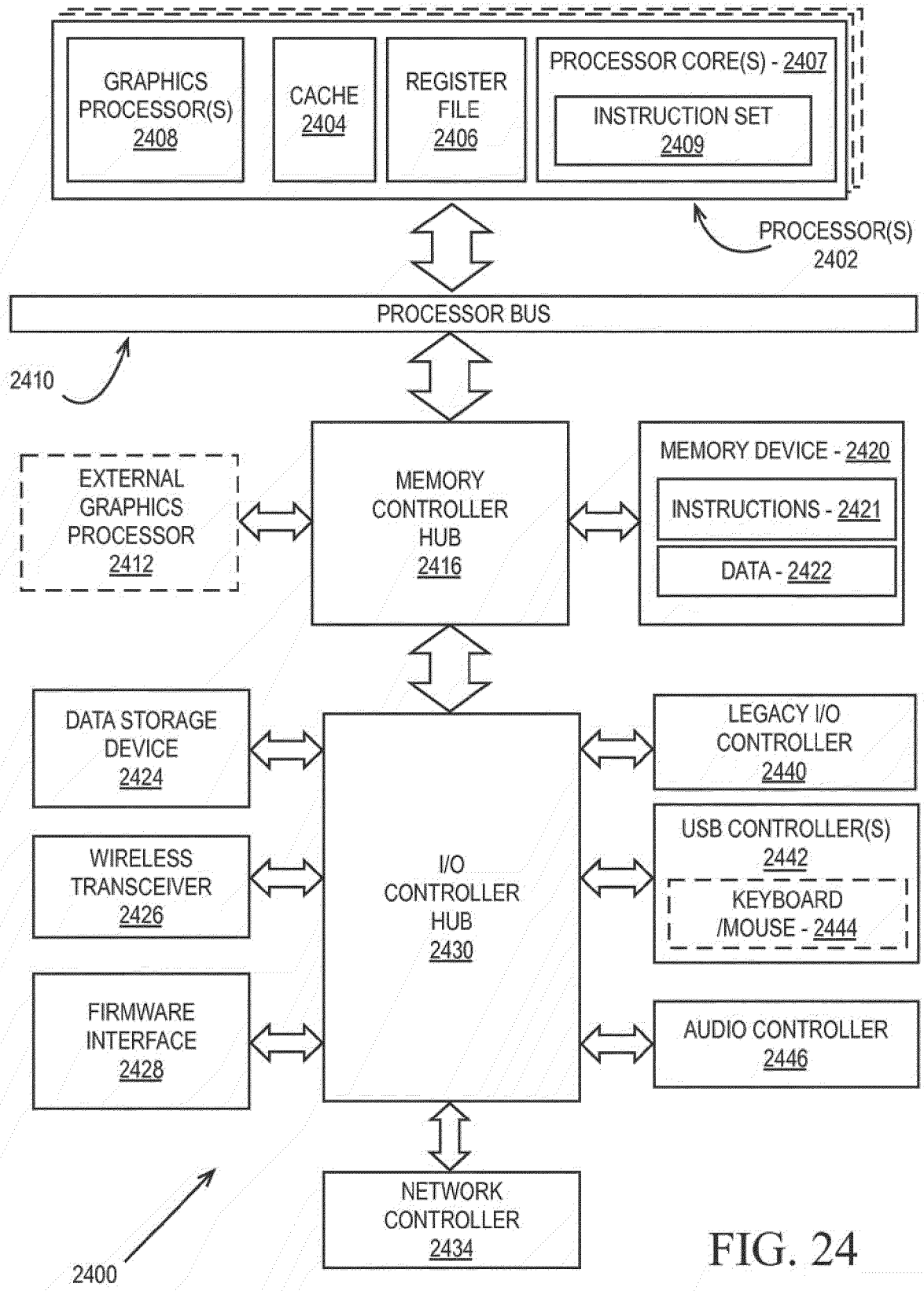

GRAPHICS PROCESSOR(S) 2408
CACHE 2404
REGISTER FILE 2406
PROCESSOR CORE(S) - 2407
INSTRUCTION SET 2409
PROCESSOR(S) 2402
PROCESSOR BUS
2410
EXTERNAL GRAPHICS PROCESSOR 2412
MEMORY CONTROLLER HUB 2416
MEMORY DEVICE - 2420
INSTRUCTIONS - 2421
DATA - 2422
DATA STORAGE DEVICE 2424
WIRELESS TRANSCEIVER 2426
FIRMWARE INTERFACE 2428
I/O CONTROLLER HUB 2430
LEGACY I/O CONTROLLER 2440
USB CONTROLLER(S) 2442
KEYBOARD /MOUSE - 2444
AUDIO CONTROLLER 2446
NETWORK CONTROLLER 2434
2400

FIG. 24

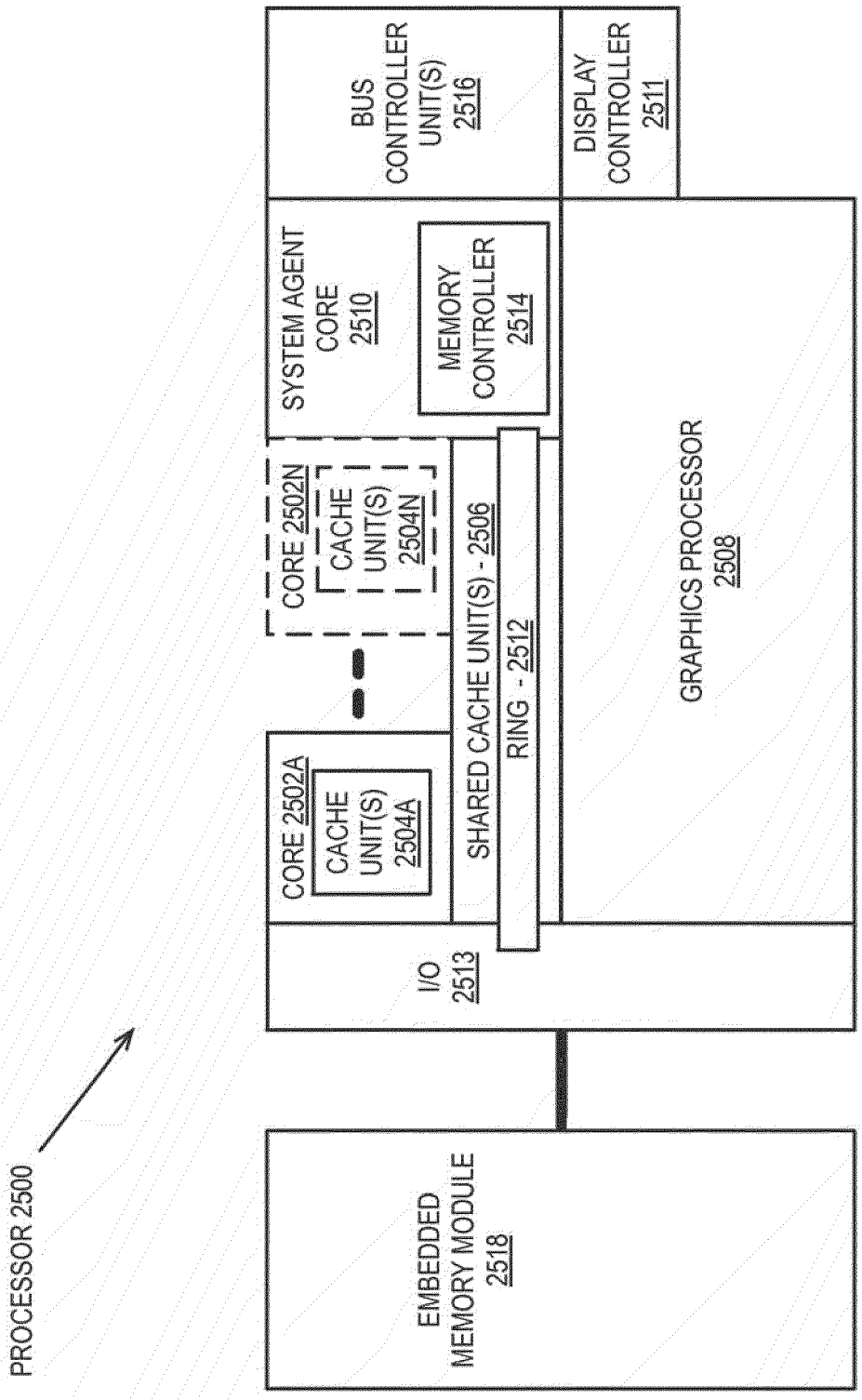

PROCESSOR 2500
EMBEDDED MEMORY MODULE 2518
I/O 2513
CORE 2502A
CACHE UNIT(S) 2504A
CORE 2502N
CACHE UNIT(S) 2504N
SHARED CACHE UNIT(S) - 2506
RING - 2512
GRAPHICS PROCESSOR 2508
SYSTEM AGENT CORE 2510
MEMORY CONTROLLER 2514
BUS CONTROLLER UNIT(S) 2516
DISPLAY CONTROLLER 2511

FIG. 25

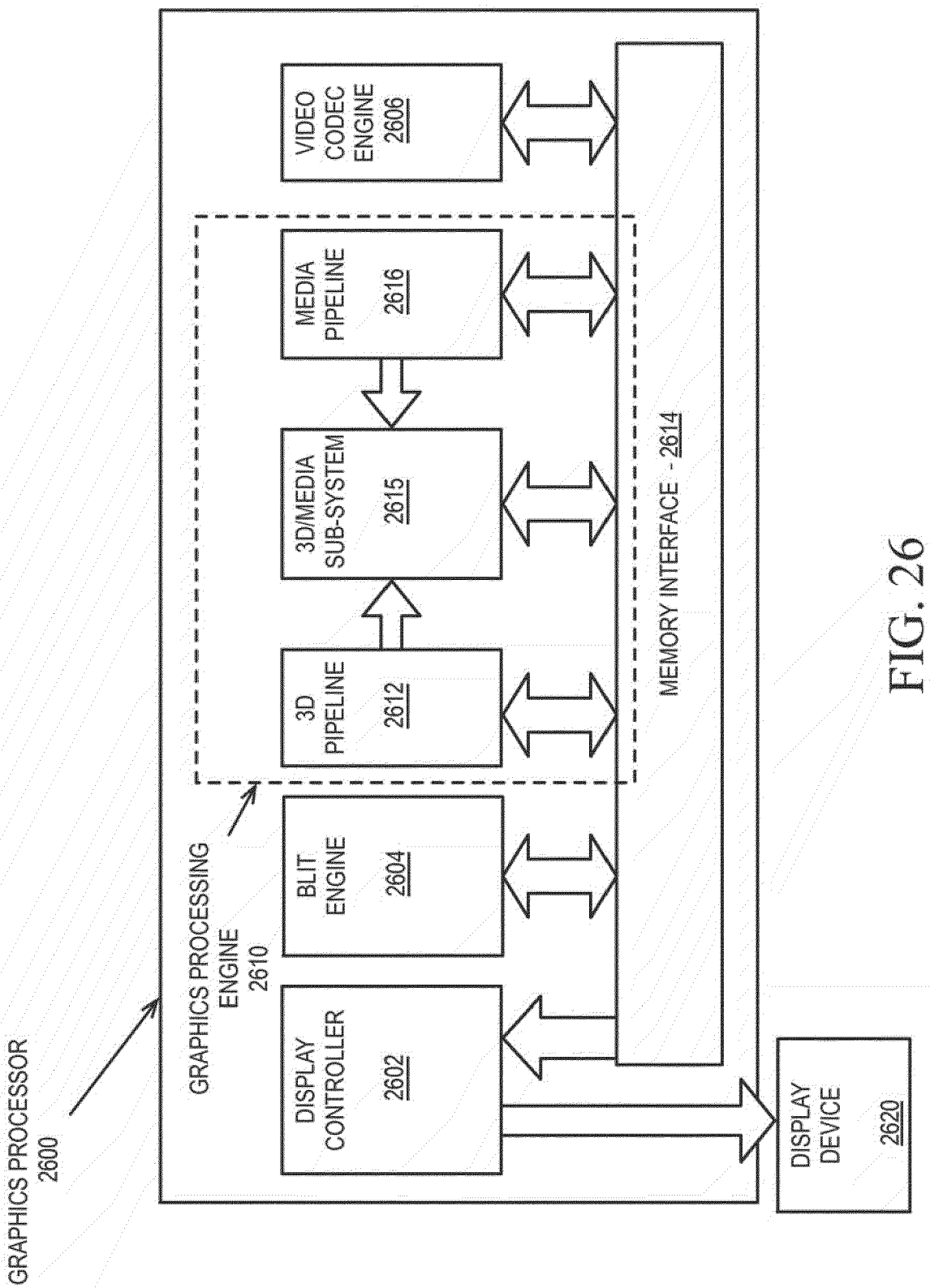

GRAPHICS PROCESSOR
2600
GRAPHICS PROCESSING ENGINE
2610
DISPLAY CONTROLLER
2602
BLIT ENGINE
2604
3D PIPELINE
2612
3D/MEDIA SUB-SYSTEM
2615
MEDIA PIPELINE
2616
VIDEO CODEC ENGINE
2606
MEMORY INTERFACE - 2614
DISPLAY DEVICE
2620

FIG. 26

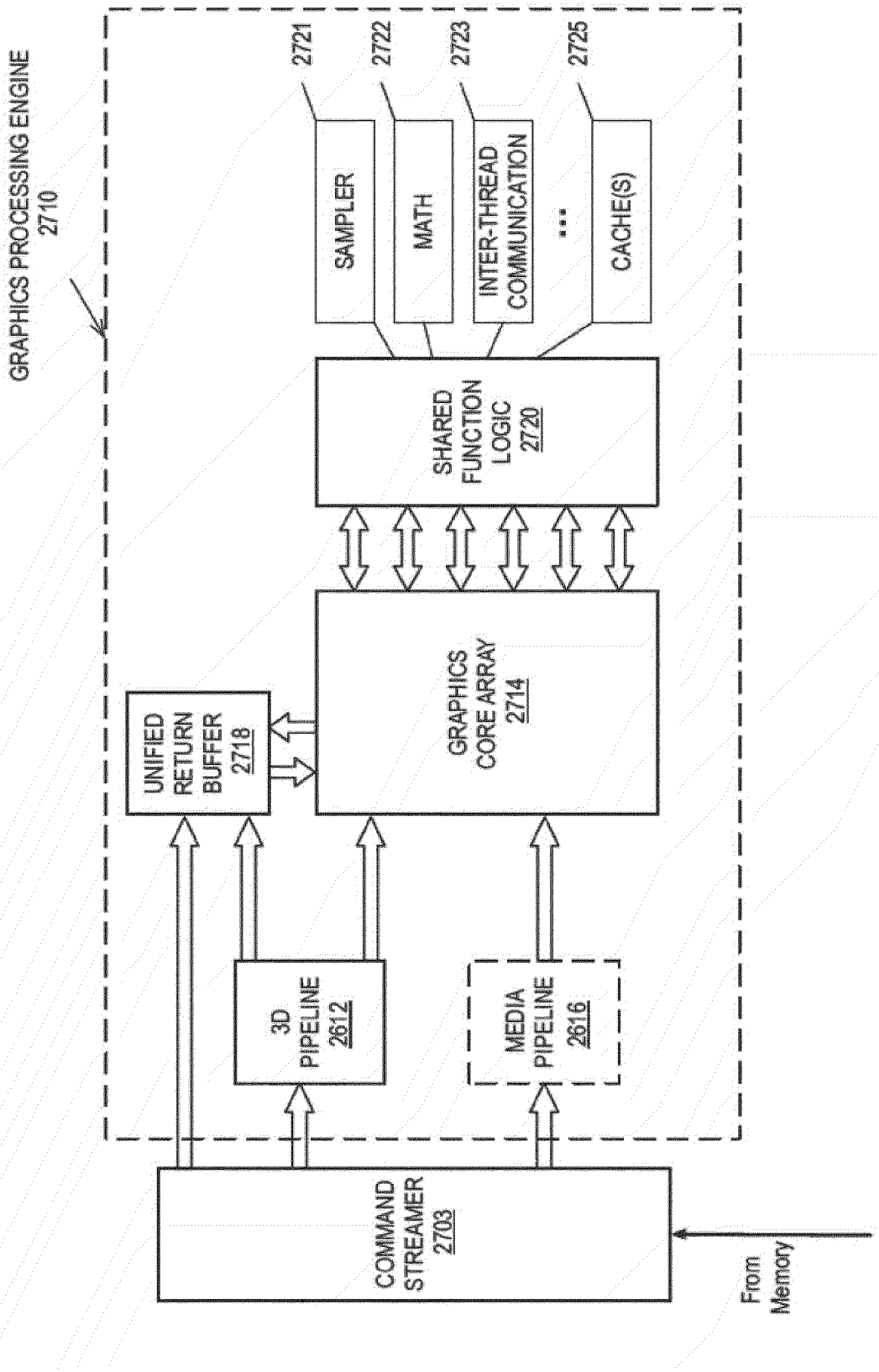

GRAPHICS PROCESSING ENGINE
2710
2721
2722
2723
2725
SAMPLER
MATH
INTER-THREAD COMMUNICATION
...
CACHE(S)
SHARED FUNCTION LOGIC 2720
GRAPHICS CORE ARRAY 2714
UNIFIED RETURN BUFFER 2718
3D PIPELINE 2612
MEDIA PIPELINE 2616
COMMAND STREAMER 2703
From Memory

FIG. 27

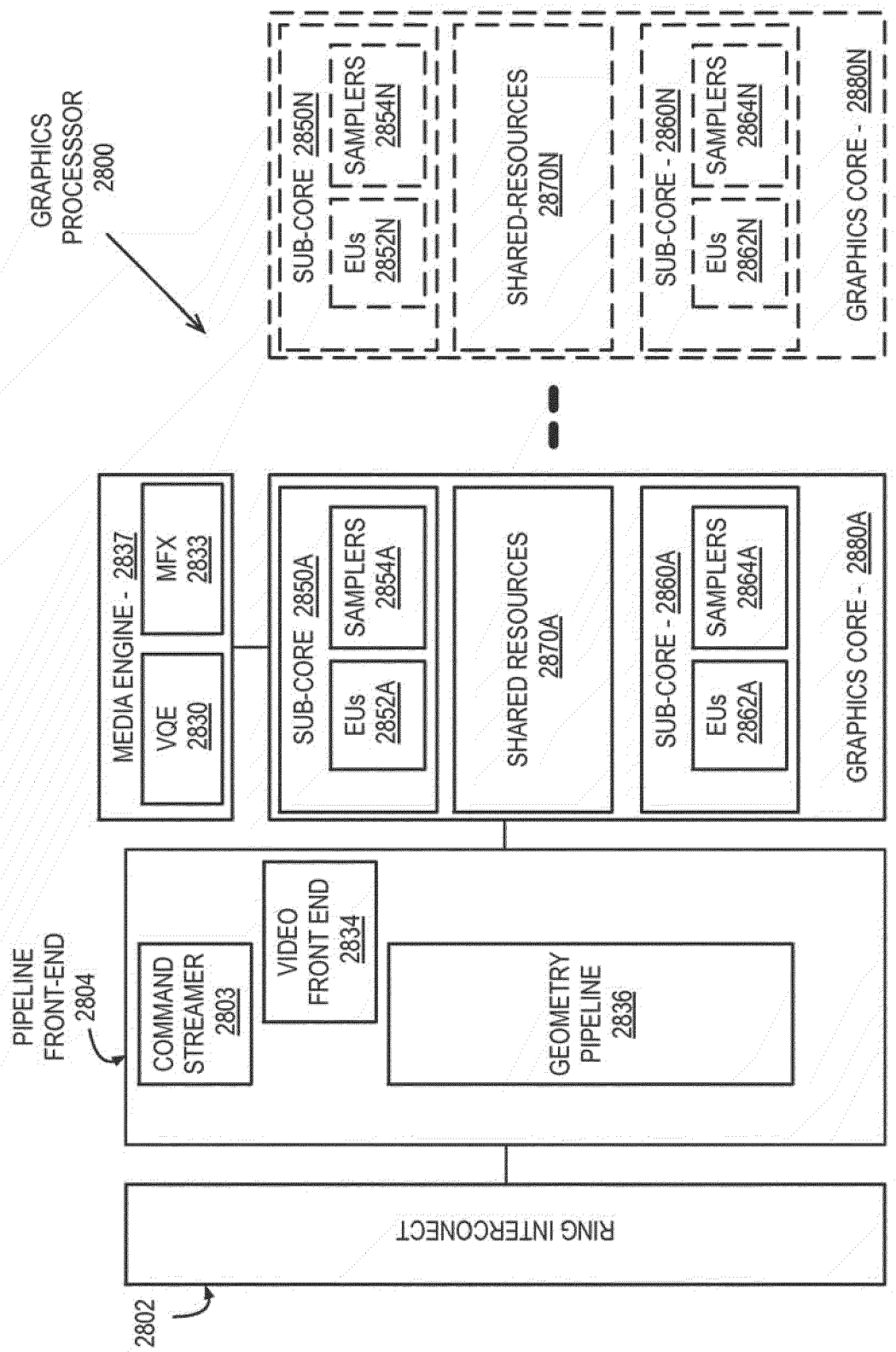

GRAPHICS PROCESSSOR 2800
SUB-CORE 2850N
EUs 2852N
SAMPLERS 2854N
SHARED-RESOURCES 2870N
SUB-CORE - 2860N
EUs 2862N
SAMPLERS 2864N
GRAPHICS CORE - 2880N
MEDIA ENGINE - 2837
VQE 2830
MFX 2833
SUB-CORE 2850A
EUs 2852A
SAMPLERS 2854A
SHARED RESOURCES 2870A
SUB-CORE - 2860A
EUs 2862A
SAMPLERS 2864A
GRAPHICS CORE - 2880A
PIPELINE FRONT-END 2804
COMMAND STREAMER 2803
VIDEO FRONT END 2834
GEOMETRY PIPELINE 2836
RING INTERCONECT
2802

FIG. 28

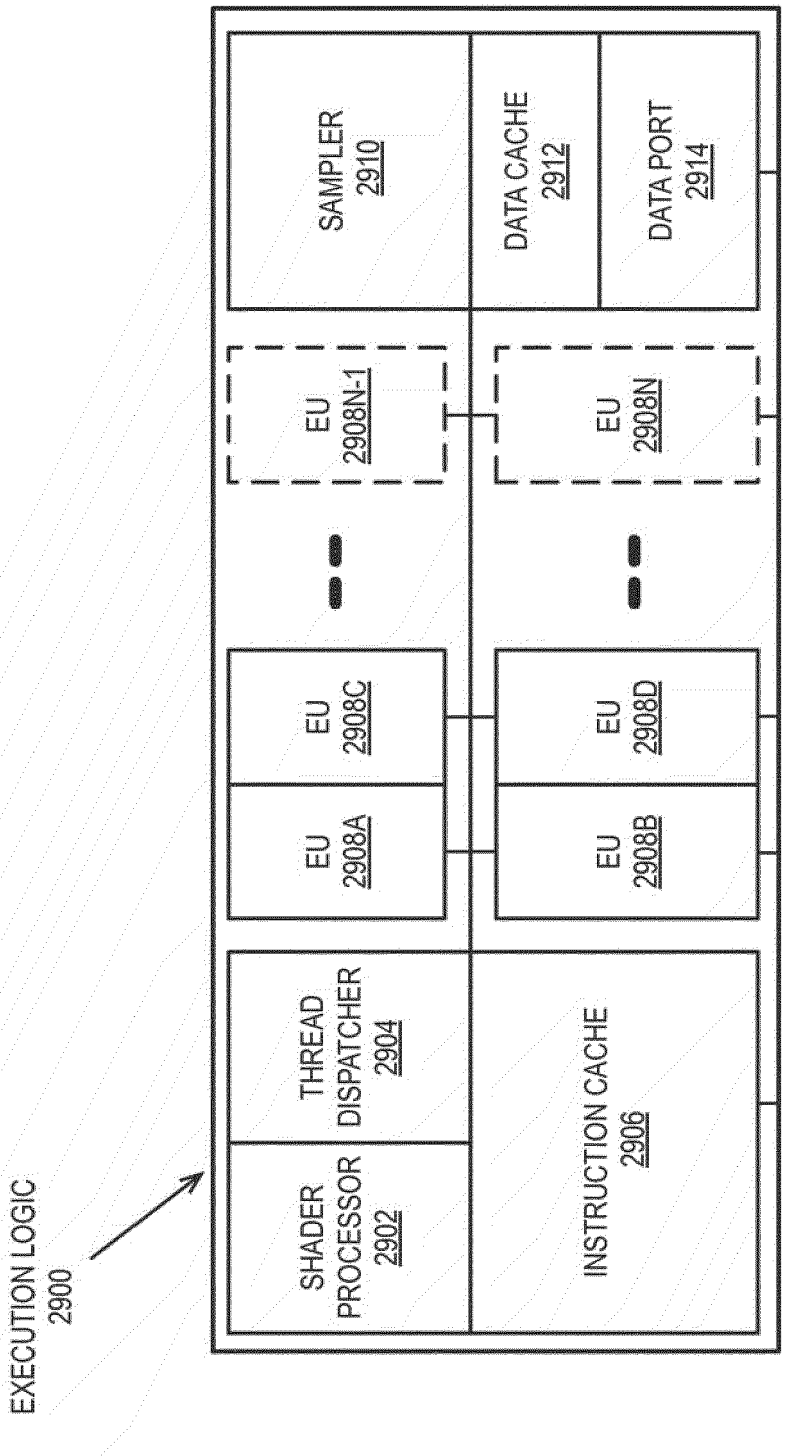

EXECUTION LOGIC
2900
SHADER PROCESSOR 2902
THREAD DISPATCHER 2904
INSTRUCTION CACHE 2906
EU 2908A
EU 2908C
EU 2908B
EU 2908D
EU 2908N-1
EU 2908N
SAMPLER 2910
DATA CACHE 2912
DATA PORT 2914

FIG. 29

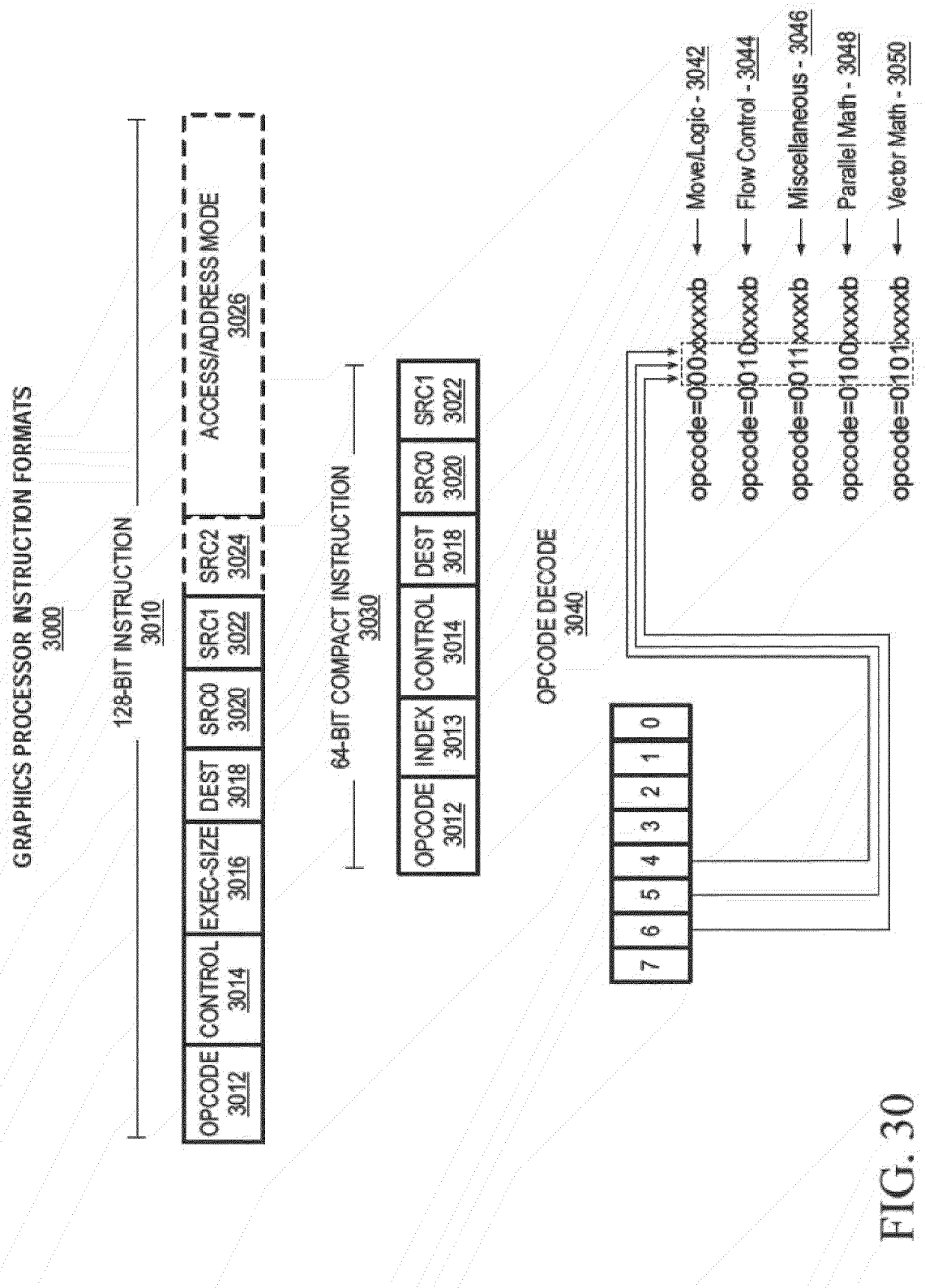

GRAPHICS PROCESSOR INSTRUCTION FORMATS
3000
128-BIT INSTRUCTION
3010
OPCODE 3012
CONTROL 3014
EXEC-SIZE 3016
DEST 3018
SRC0 3020
SRC1 3022
SRC2 3024
ACCESS/ADDRESS MODE 3026
64-BIT COMPACT INSTRUCTION
3030
OPCODE 3012
INDEX 3013
CONTROL 3014
DEST 3018
SRC0 3020
SRC1 3022
OPCODE DECODE
3040
7 6 5 4 3 2 1 0
opcode=000xxxxb
opcode=001xxxxb
opcode=010xxxxb
opcode=011xxxxb
opcode=100xxxxb
opcode=101xxxxb
Move/Logic - 3042
Flow Control - 3044
Miscellaneous - 3046
Parallel Math - 3048
Vector Math - 3050

FIG. 30

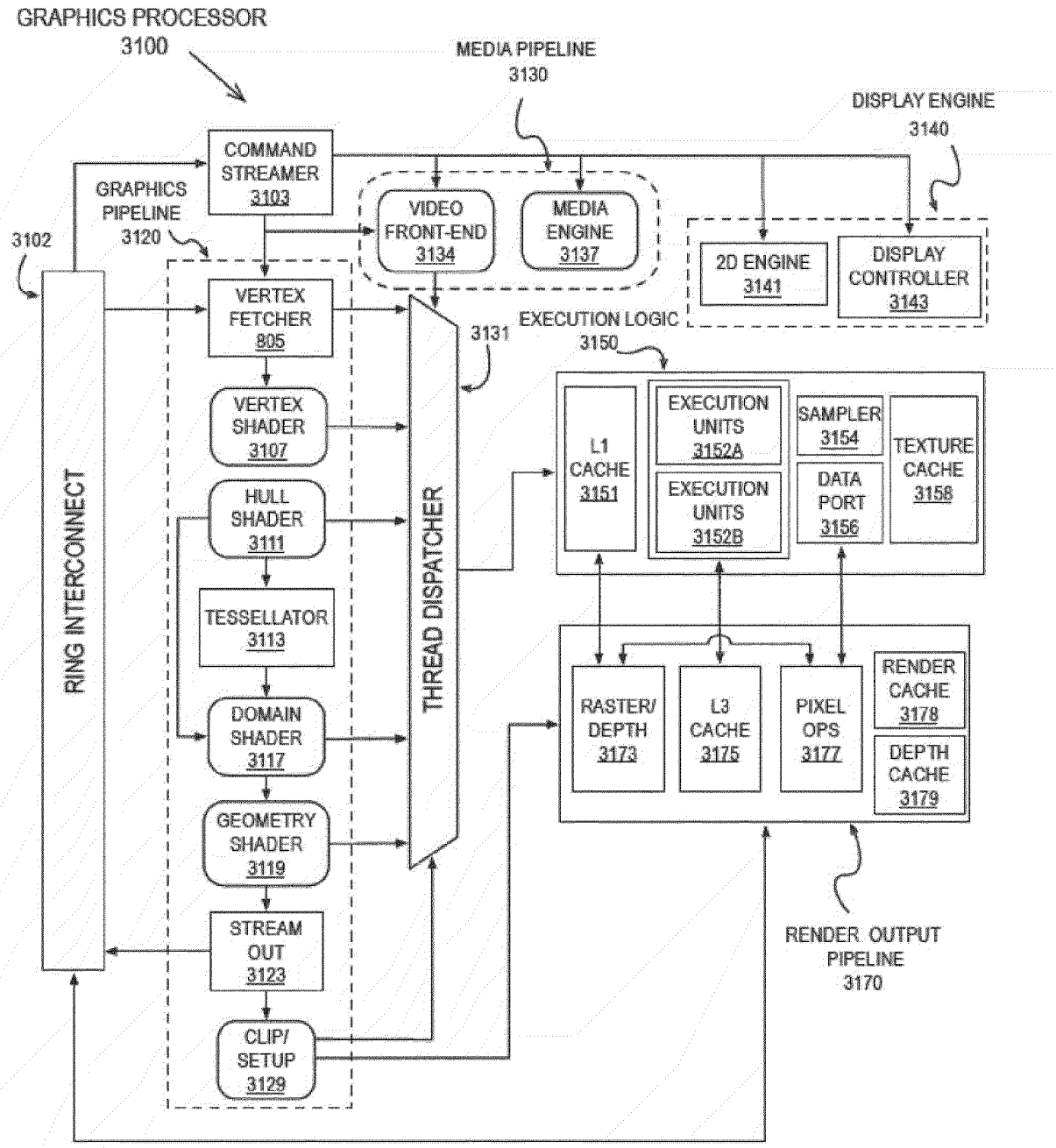

GRAPHICS PROCESSOR
3100
MEDIA PIPELINE
3130
DISPLAY ENGINE
3140
COMMAND STREAMER
3103
GRAPHICS PIPELINE
3120
3102
VIDEO FRONT-END
3134
MEDIA ENGINE
3137
2D ENGINE
3141
DISPLAY CONTROLLER
3143
VERTEX FETCHER
805
VERTEX SHADER
3107
HULL SHADER
3111
TESSELLATOR
3113
DOMAIN SHADER
3117
GEOMETRY SHADER
3119
STREAM OUT
3123
CLIP/ SETUP
3129
RING INTERCONNECT
THREAD DISPATCHER
3131
EXECUTION LOGIC
3150
L1 CACHE
3151
EXECUTION UNITS
3152A
EXECUTION UNITS
3152B
SAMPLER
3154
DATA PORT
3156
TEXTURE CACHE
3158
RASTER/ DEPTH
3173
L3 CACHE
3175
PIXEL OPS
3177
RENDER CACHE
3178
DEPTH CACHE
3179
RENDER OUTPUT PIPELINE
3170

FIG. 31

FIG. 32A GRAPHICS PROCESSOR COMMAND FORMAT
3200

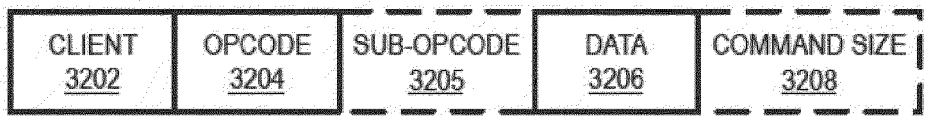
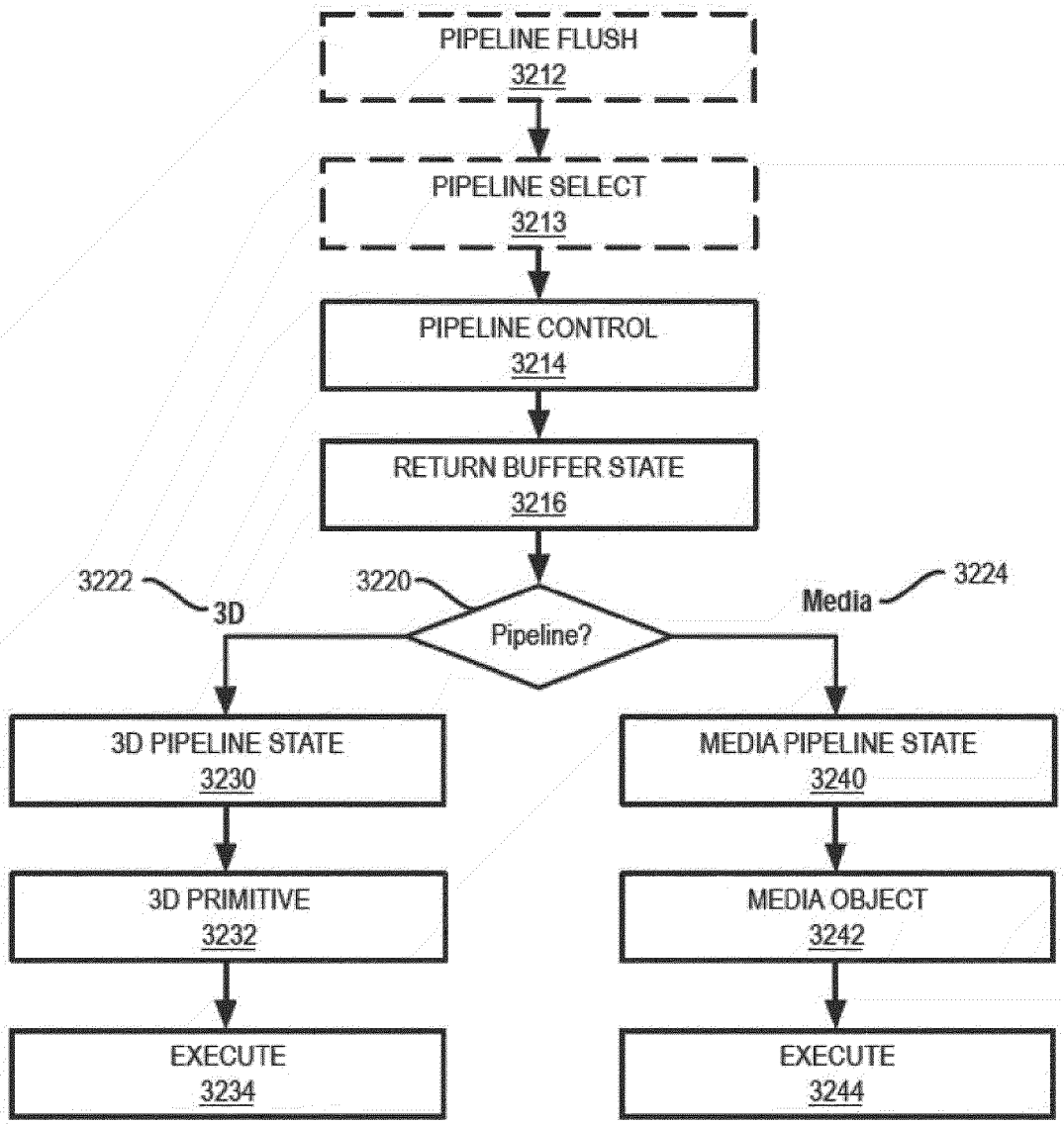


FIG. 32B GRAPHICS PROCESSOR COMMAND SEQUENCE
3210



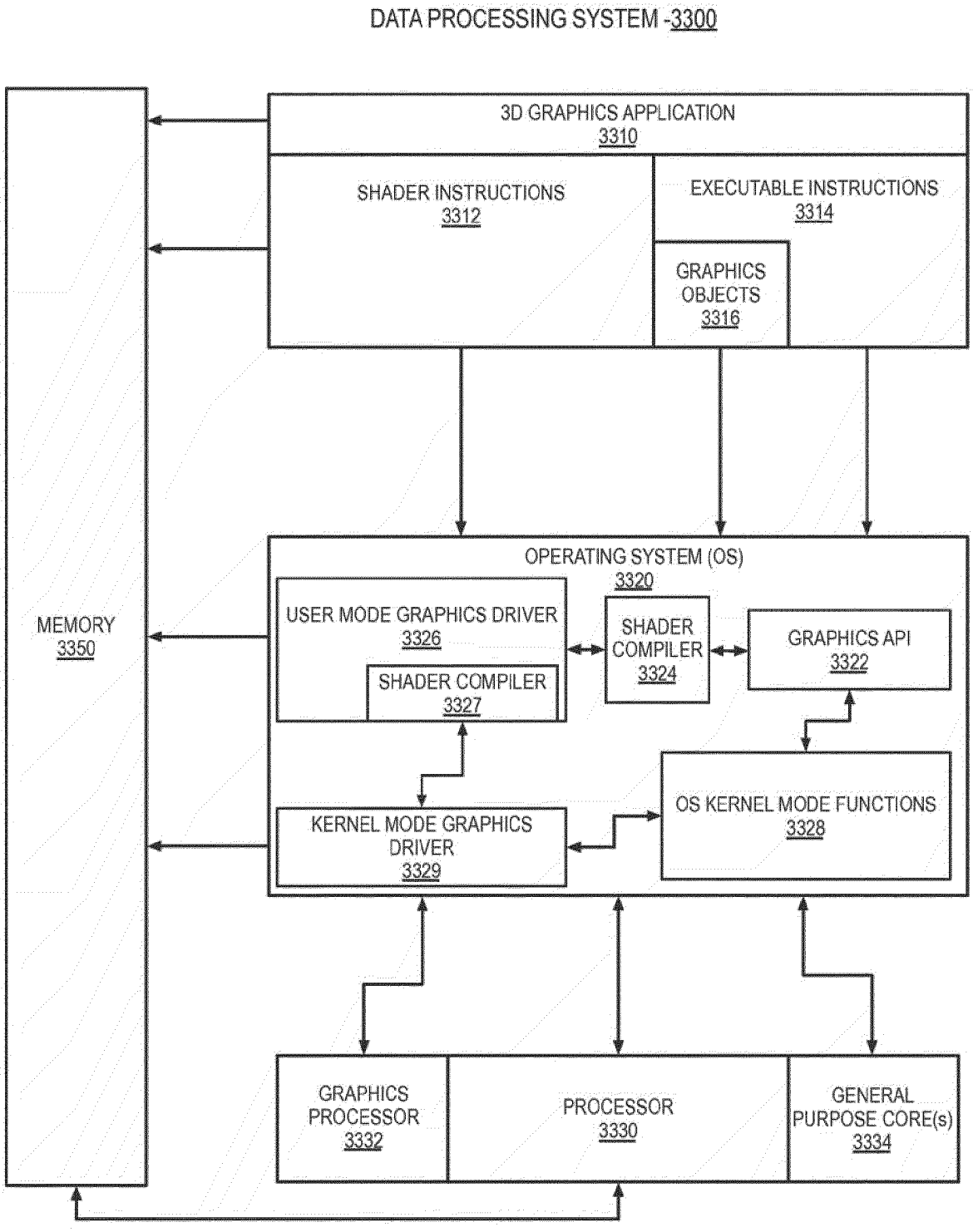

DATA PROCESSING SYSTEM -3300
3D GRAPHICS APPLICATION
3310
SHADER INSTRUCTIONS
3312
EXECUTABLE INSTRUCTIONS
3314
GRAPHICS OBJECTS
3316
MEMORY
3350
OPERATING SYSTEM (OS)
3320
USER MODE GRAPHICS DRIVER
3326
SHADER COMPILER
3327
SHADER COMPILER
3324
GRAPHICS API
3322
KERNEL MODE GRAPHICS DRIVER
3329
OS KERNEL MODE FUNCTIONS
3328
GRAPHICS PROCESSOR
3332
PROCESSOR
3330
GENERAL PURPOSE CORE(s)
3334

FIG. 33

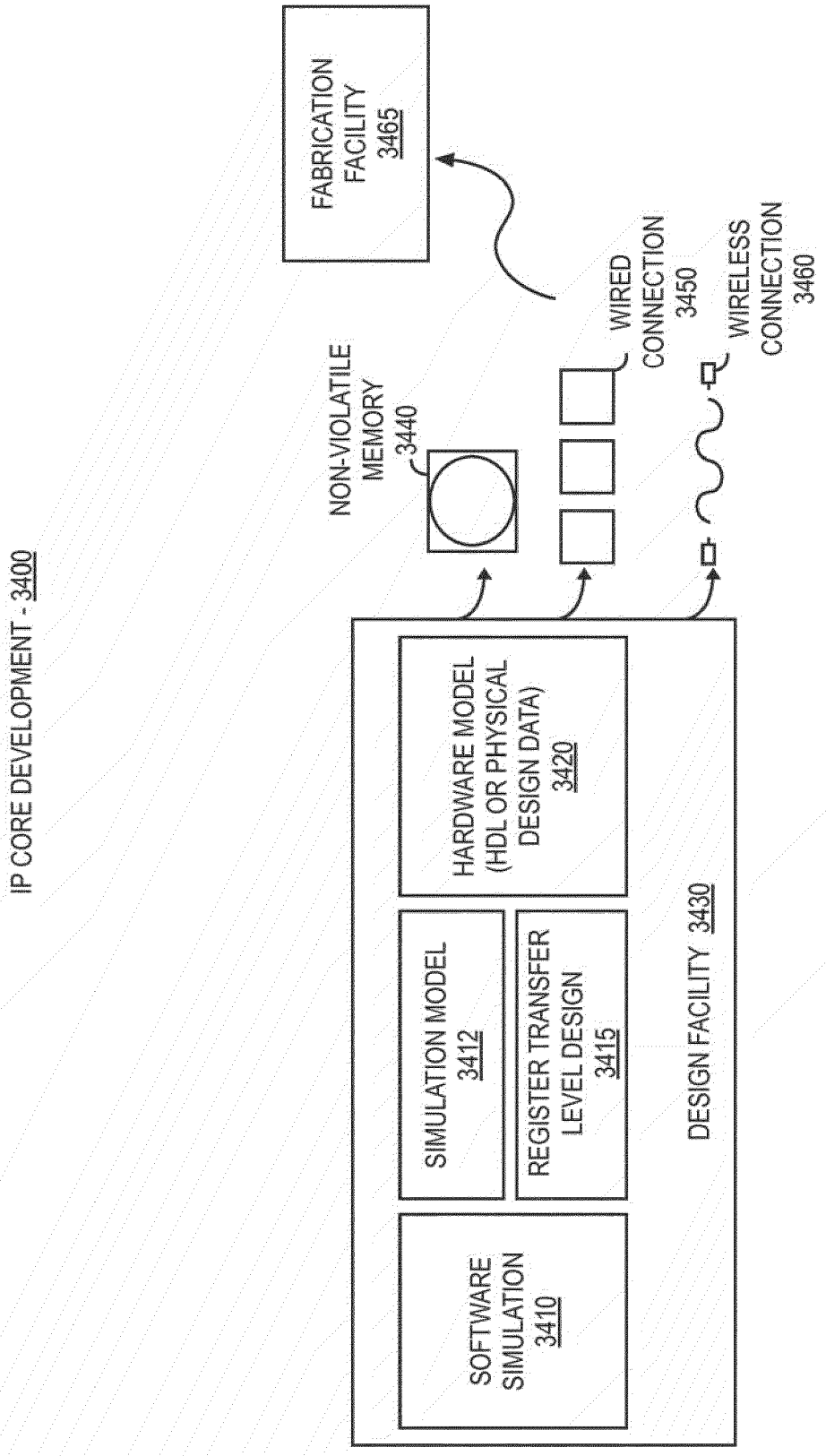

IP CORE DEVELOPMENT - 3400
FABRICATION FACILITY 3465
NON-VIOLATILE MEMORY 3440
WIRED CONNECTION 3450
WIRELESS CONNECTION 3460
SOFTWARE SIMULATION 3410
SIMULATION MODEL 3412
REGISTER TRANSFER LEVEL DESIGN 3415
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 3420
DESIGN FACILITY 3430

FIG. 34

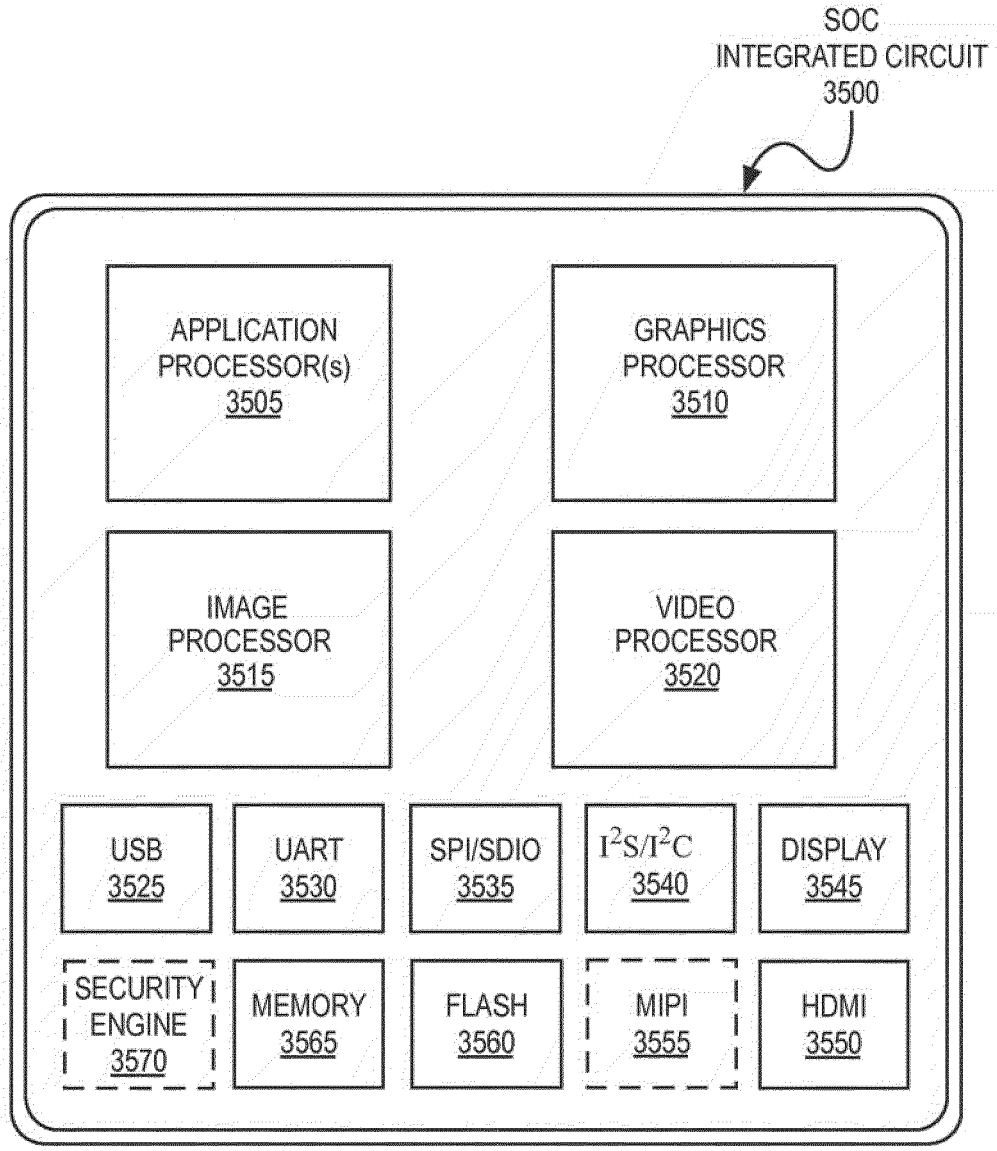

SOC
INTEGRATED CIRCUIT
3500
APPLICATION PROCESSOR(s) 3505
GRAPHICS PROCESSOR 3510
IMAGE PROCESSOR 3515
VIDEO PROCESSOR 3520
USB 3525
UART 3530
SPI/SDIO 3535
I2S/I2C 3540
DISPLAY 3545
SECURITY ENGINE 3570
MEMORY 3565
FLASH 3560
MIPI 3555
HDMI 3550

FIG. 35

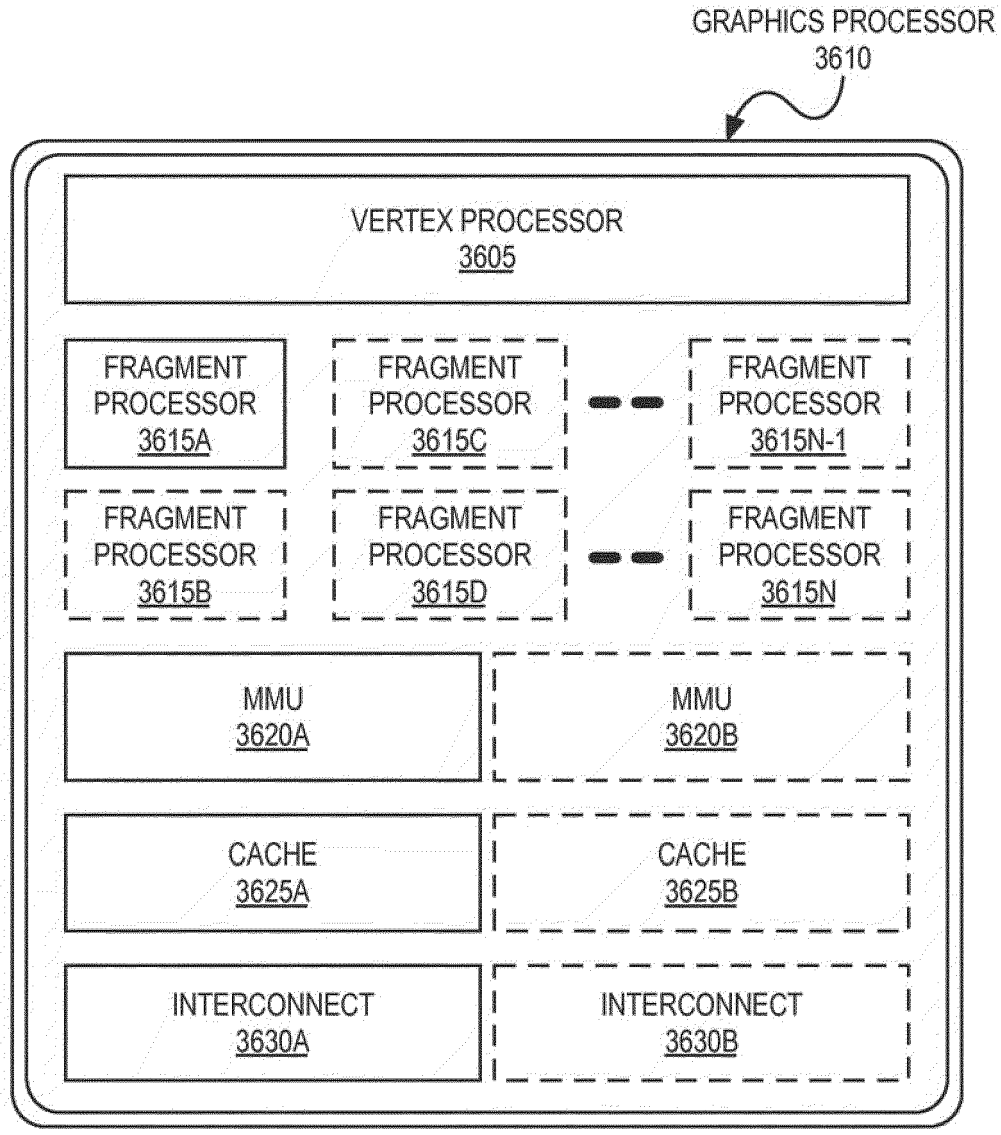

GRAPHICS PROCESSOR
3610
VERTEX PROCESSOR
3605
FRAGMENT PROCESSOR
3615A
FRAGMENT PROCESSOR
3615C
FRAGMENT PROCESSOR
3615N-1
FRAGMENT PROCESSOR
3615B
FRAGMENT PROCESSOR
3615D
FRAGMENT PROCESSOR
3615N
MMU
3620A
MMU
3620B
CACHE
3625A
CACHE
3625B
INTERCONNECT
3630A
INTERCONNECT
3630B

FIG. 36

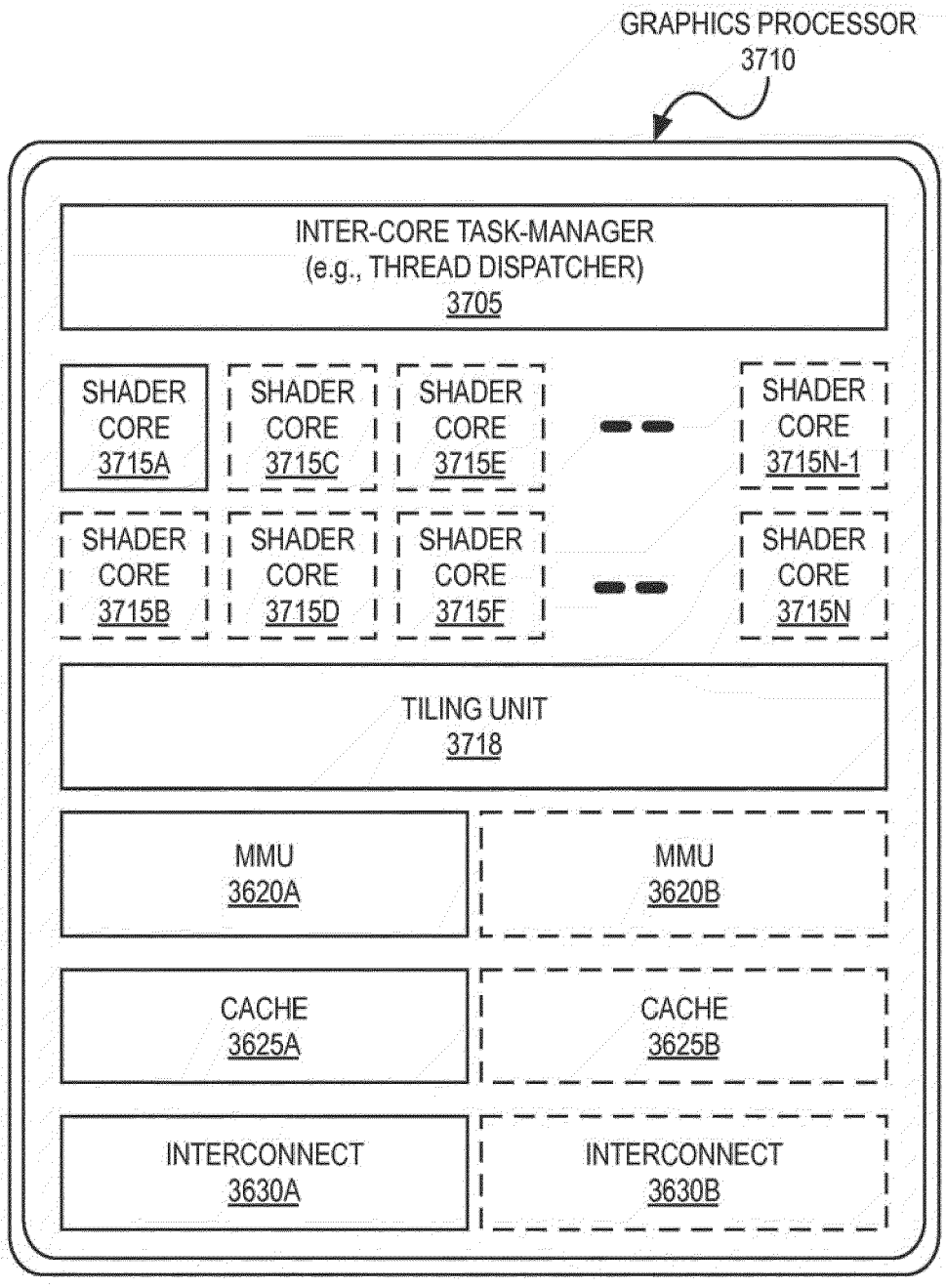

GRAPHICS PROCESSOR
3710
INTER-CORE TASK-MANAGER
(e.g., THREAD DISPATCHER)
3705
SHADER CORE 3715A
SHADER CORE 3715C
SHADER CORE 3715E
SHADER CORE 3715N-1
SHADER CORE 3715B
SHADER CORE 3715D
SHADER CORE 3715F
SHADER CORE 3715N
TILING UNIT
3718
MMU
3620A
MMU
3620B
CACHE
3625A
CACHE
3625B
INTERCONNECT
3630A
INTERCONNECT
3630B

FIG. 37

Europäisches Patentamt
European Patent Office
Office européen des brevets

EUROPEAN SEARCH REPORT

Application Number
EP 21 16 5109

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (IPC)
X	WO 2017/049496 A1 (INTEL CORP [US]; YANG YI [CN] ET AL.) 30 March 2017 (2017-03-30) * paragraph [0025] - paragraph [0035]; figures 1, 2 * * paragraph [00115] - paragraph [00136]; figure 13B *	1-11	INV. G06F7/483 G06F7/544 G06F9/30 G06N3/04 G06N3/063 G06N3/08
A	US 2016/328647 A1 (LIN DEXU [US] ET AL) 10 November 2016 (2016-11-10) * paragraph [0025] - paragraph [0046] *	1-11	
A	MATTHIEU COURBARIAUX ET AL: "TRAINING DEEP NEURAL NETWORKS WITH LOW PRECISION MULTIPLICATIONS", CORR (ARXIV), 23 October 2015 (2015-10-23), pages 1-10, XP055302535, * Sections 3, 4, 5 *	1-11	

TECHNICAL FIELDS SEARCHED (IPC)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
Munich	22 June 2021	Tenbieg, Christoph

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT ON EUROPEAN PATENT APPLICATION NO. EP 21 16 5109

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report. The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

22-06-2021

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2017049496 A1	30-03-2017	US 2019073582 A1	07-03-2019
		WO 2017049496 A1	30-03-2017
US 2016328647 A1	10-11-2016	CN 107646116 A	30-01-2018
		EP 3295382 A1	21-03-2018
		EP 3852016 A1	21-07-2021
		US 2016328647 A1	10-11-2016
		WO 2016182659 A1	17-11-2016

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82

REFERENCES CITED IN THE DESCRIPTION

This list of references cited by the applicant is for the reader's convenience only. It does not form part of the European patent document. Even though great care has been taken in compiling the references, errors or omissions cannot be excluded and the EPO disclaims all liability in this regard.

Non-patent literature cited in the description

- **SHANE COOK.** CUDA Programming. 2013, 37-51 **[0003]**
- A Comprehensive Guide to GPU Programming. **NICHOLAS WILT.** CUDA Handbook. June 2013 **[0003]**